PATH PLANNING SOFTWARE AND GRAPHICS INTERFACE FOR AN AUTONOMOUS VEHICLE, ACCOUNTING FOR TERRAIN FEATURES

By

VLAD HUREZEANU

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2000

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2000		**00-00-2000 to 00-00-2000**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Path Planning Software and Graphics Interface for an Autonomous Vehicle, Accounting for Terrain Features	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Center for Intelligent Machines and Robotics,Department of Mechanical Engineering,University of Florida,Gainesville,FL,32611**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**134**	

ACKNOWLEDGMENTS

The author would like to express his thanks to the members of his supervisory committee, Dr. Carl Crane, Dr. Joseph Duffy, and Dr. Paul Mason, for their contribution and support. Special recognition goes to Dr. Carl Crane who gave the author the opportunity to continue his education in a field of his desire. Also, the author wishes to express his thanks to Dr. Carl Crane for his advice and for his mentoring on this thesis.

This work would not have been possible without the help of Jeff Witt and David Novick, whose invaluable advice got the author on the right path of writing software as well as thinking like an engineer. Also, thanks go to David Armstrong whose guidance has helped the author understand the concept of an autonomous vehicle. Thanks are expressed as well to Arfath Pasha whose software has made the author's work easier.

The support of the Air Force Research Laboratory at Tyndall Air Force Base is gratefully acknowledged.

Finally, special thanks go to my mother, Mihaela Barrone, who has offered the moral support throughout the development of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

PATH PLANNING SOFTWARE AND GRAPHICS INTERFACE
FOR AN AUTONOMOUS VEHICLE, ACCOUNTING
FOR TERRAIN FEATURES

By

Vlad Hurezeanu

December 2000

Chairman: Dr. Carl D. Crane III
Major Department: Mechanical Engineering

A Navigation Test Vehicle (NTV) is being developed at the Center for Intelligent Machines and Robots at the University of Florida under the sponsorship of the Air Force Research Laboratory at Tyndall Air Force Base. This vehicle performs tasks to include surveying fields, laying mines, and teleoperation. The capability of the vehicle will be increased if its supporting software plans paths that take into account the terrain features.

The objective of this work is to plan a survey path for a specified region taking terrain elevation data into account. A survey path is defined as a path that provides complete coverage of a defined polygonal region. This work is an extension of previous work that assumed that the region to be surveyed was flat. Now the characteristics of the vehicle, specifically the maximum allowable pitch and roll, will be accounted for when planning the path.

Data that contains elevation points are available from government databases. Using three different programs the data are transformed from the vector format in which it is found to a Virtual Reality Modeling Language format in order to be viewed with a common web browser. Data are stored into a grid of $m \times n$ elements and the user selects the coordinates of the region that will be surveyed. The previous 2D algorithm is then executed in which the primary direction of the survey varies from zero to 180 degrees. For each path a cost is computed in which the cost is based on pitch and roll knowing the maximum that the vehicle can withstand. For the NTV the maximum roll and pitch angle is 15^{o}. If the pitch or roll is larger than 15 degrees a very high cost is associated with that part of the path. After each path is processed, the path with overall minimum cost is written to a file. If the minimum path contains any slopes that are inaccessible they are recorded as obstacles. The last step is an obstacle avoidance algorithm, which generates small sub-paths around the inaccessible regions and ultimately provides the safest and least expensive path for surveying the given field. To prove the reliability of the algorithm, multiple cases have been taken into account such as obstacles on the boundaries, overlapping obstacles, as well as large non-traversable regions. The algorithm has been implemented and tested on a Silicon Graphics workstation and on the NTV.

CHAPTER 1
INTRODUCTION

Problem Statement

Research in the area of autonomously navigating vehicles has been ongoing at the Center for Intelligent Machines and Robotics since approximately 1990. Although there are many topics involved in the functionality of such a vehicle, path planning is the main goal of this research.

Path planning deals with computing a way of achieving a desired goal position starting at a specified position. There are two different types of path planning: goal-to-goal and survey. The research presented in this work deals exclusively with survey planning and it accounts for two-dimensional and three-dimensional planning. Although three-dimensional planning assumes movement and rotation about three distinct axes we will still call it three-dimensional, even if there is no such activity in this case.

This work focuses on designing path planning software that takes into account the terrain features as they relate to the capabilities of the vehicle.

The inputs for this task are the following:

1. The latitude and longitude coordinates of the vertices of a polygon that represents the region to be surveyed.
2. An ordered set of data for the terrain, that contains information about latitude, longitude, and altitude for points within the polygon to be surveyed. The terrain data will be stored in an evenly spaced grid.

3. The distance between adjacent surveying lines for the planned path, i.e. the resolution of the survey.
4. The locations of obstacles located within the area to be surveyed. These obstacles are defined by a polygon whose latitude and longitude vertice coordinates are specified.
5. The maximum allowable pitch angle and roll angle for the vehicle.
6. The length and width of the vehicle performing the survey.
7. The minimum allowable radius of curvature for the vehicle.

Based on the above inputs the objective was to determine an efficient survey path that would achieve a complete survey of the area without violating vehicle constraints while also avoiding known obstacles.

Background

The Navigation Test Vehicle (NTV) (Figure 1.1) is one of a series of autonomous vehicles that have been developed for the Air Force Research Laboratory at Tyndall Air Force Base, Florida. Research has focused in five areas, i.e.: path planning, positioning systems, motion execution, obstacle avoidance, and system architecture. As previously stated, this paper deals exclusively with the path planning problem but a brief overview of the overall vehicle architecture is presented in Chapter 5.

Figure 1.1. Navigation Test Vehicle (NTV)

A first version of the two-dimensional survey path planning algorithm was developed by Ran93. The author improved this software by organizing it in a modular structure. The three-dimensional work, which is presented in this research, is an extension of the two-dimensional path planning and creates an efficient way of surveying fields.

CHAPTER 2
REVIEW OF LITERATURE

The author has performed an extensive search of the available resources in order to find material related to the three-dimensional survey path planning. However, there was nothing found that showed other methods of survey path planning. All the encountered articles were related to goal-to-goal path planning. Since there are a lot of resources regarding the two-dimensional goal-to-goal path planning only the three-dimensional goal-to-goal path planning aspect will be analyzed in this paper in order to determine if any of the methods encountered are similar to the ones used in this paper. In particular there are numerous algorithms that deal with finding the best path or finding the optimal map. However, the author will describe the algorithms that effectively achieve complicated task of three-dimensional path planning. Path planning can be divided into two major categories: on-line path planning and off-line path planning. Even though the purpose of this paper is pure off-line path planning, one example of an on-line path planning algorithm will be discussed here.

On-Line Path Planning

On-line path planning is a set of algorithms that plan and execute a path at the same time. This task is based on feedback from a variety of sensors such as sonar and laser. Every task, like obstacle avoidance and backing, is analyzed based on data obtained from these sensors.

Stenntz and his colleagues (Ste93) at Carnegie Mellon University developed a completely self-contained vehicle called NavLab II. It uses the on-line path planning strategy by developing digital maps of the terrain while traveling. To achieve safe traveling, the vehicle scans the area in front of it, plans a path, and follows it. The path planner handles two tasks. The first task, which is called the spatial phase, plans a path by using heuristics. An initial path called a global path is designed to take the vehicle from start to goal. Along this path there are numerous control points that divide the global path in smaller segments. Once one control point is reached a curve is fit from it to the next control point. A simulation describes the vehicle's movement on the curve. If the path is unsafe or it has obstacles, two new paths are computed. The computer selects from one of the new paths and the cycle repeats itself until a safe final path is found. The second task, which is called the temporal phase, assigns speeds to different trajectory points based on the dynamics and kinematics of the vehicle.

Off-Line Path Planning

Off-line path planning is a set of algorithms that process the existing data before the vehicle starts moving. In this way the user can visualize the path and detect possible errors in the algorithm. This method provides the safest possible path planning

Rowe and Alexander (Row00) presented a method for finding optimal-path maps for a specific terrain by partitioning the field into areas with the same optimal-path performance. Even though this method is applied to two-dimensional space, it is presented here because it uses weighted regions, which are done at a 100-meter resolution and which are developed by the U.S. Defense Mapping Agency. The authors describe

two known methods of approximating optimal-path maps: pre-computing a set of optimal-maps and wavefront-propagation on a uniform grid, as well as a new algorithm also called continuous-Dijkstra. A cost, C, and a behavior, B (P), are associated with each optimal path. Having these inputs as well as the start and goal points the algorithm follows Snell's law at each edge. Wherever the behavior changes the respective region is partitioned into subspaces called wedges. This partition is done recursively until each weighted region segment corresponding to each part of each wedge has the same behavior.

Rowe (Row90a) has chosen the same homogeneous-cost-per-distance or weighted region method to account for roads, rivers, and obstacles. The author's main principle is called Shortcut Meta-Heuristic: a shortcut across a turn in the optimal-path map is impossible and much more expensive compared with the turn itself. Initially the algorithm models the roads, rivers, and obstacles as line segments. Then all optimal shortcuts between these line segments are chosen and an ellipse is formed by the start and goal points. After the edges lying outside the ellipse are thrown away, a weighted graph search is applied by using the A* search algorithm. A penalty is applied if the path crosses rivers.

Rowe (Row97) also presents a new theory regarding anisotropism in path planning problems. He argues that trying to apply qualitative modeling involves a lot of discontinuities that prevent the use of the calculus of variations. His method would divide such a problem into sub-goals without discontinuities, which would be much easier optimized. His paper deals with anisotropism generated by gravity, friction, winds and currents, directional danger, maximum power, and overturn danger but it ignores

issues such as obstacle avoidance and isotropic varying costs. He also argues that conventional wave front propagation algorithms are not proper for this kind of problem because they offer a fixed set of solutions, which might not be optimal. His solution also includes ray-tracing methods.

Rowe and Kanayama (Row94) present a solution to finding near-optimal paths on the surface of a cone. Their approach takes into account gravity and friction but ignores the size of the vehicle. They prove that by using a cone as the surface they found 22 behavioral paths, in comparison with a polyhedral surface, which generates only four paths. They use discontinuities in their model, which makes it opposite to the method that was described by Rowe (Row97).

Rowe and Ross (Row90c) present yet another thesis regarding the anisotropic (heading-dependent) related problems in path-planning algorithms. They use polyhedral approximation of the terrain instead of the classical methods: imposition on a uniform grid and average effects in different directions. The authors argue that grid-based solutions can have the following disadvantages: wasted computation, wasted space, path-cost errors due to directional biases, round off errors accumulation, and inability of finding a solution when the grid size is increased. To prove their theory they used the idea that there are only four ways to travel an anisotropic field: straight across without braking, straight across without braking but as close as possible to the unallowed paths, avoiding possible obstacles along the path, and straight across with braking. They also prove that their algorithm is computer time efficient. In their work they consider a smaller vehicle than the path and they consider it as a two-dimensional problem.

They construct a model where they apply the laws of physics concerning gravity and friction. To this model, the following issues are being applied: anisotropic phenomena with braking, error analysis of the cost calculations, time-minimizing paths, and anisotropic-obstacle phenomena. The directions that they consider for their model at any given point in the terrain are presented in Figure 2.1.

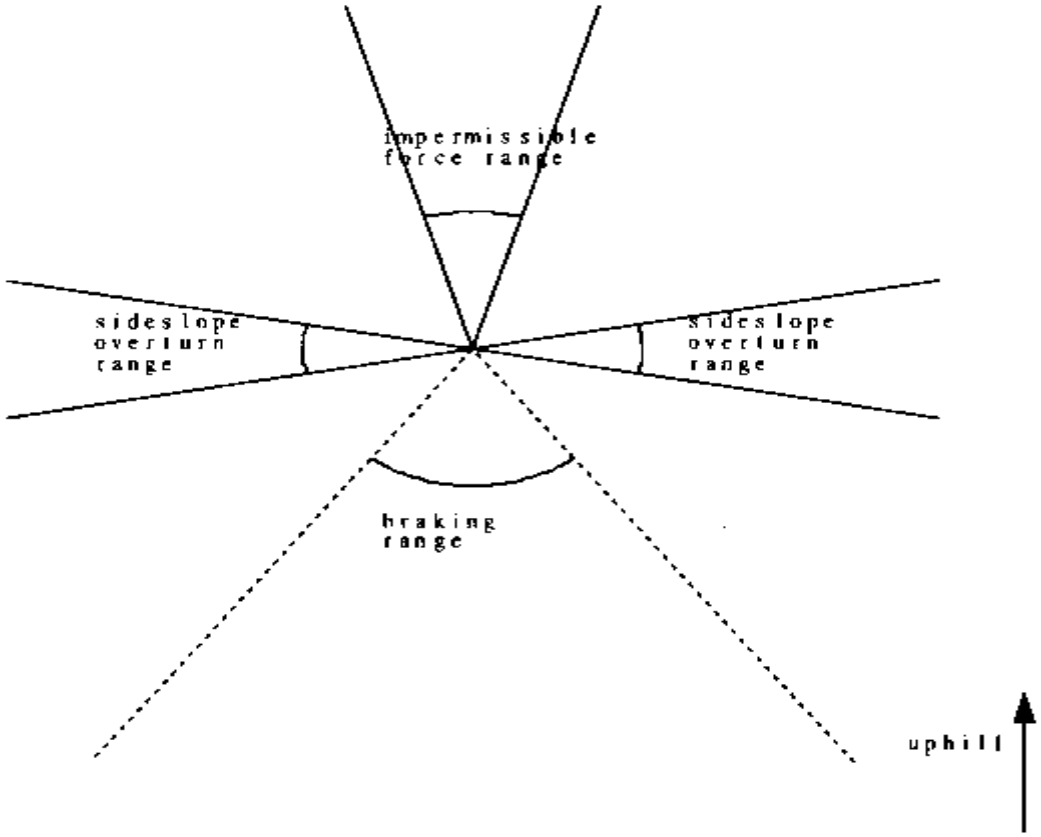

Figure 2.1. Impermissible and braking heading ranges at a point

The authors present four types of anisotropic traversals and they create lemmas and theorems for each one of them. Isotropic obstacles, or regions where travel is not permitted in any direction are also considered. The last step in the algorithm is to apply the A^* search. They also present a four-step method for polyhedral approximation of

terrain, which is required for their model. As a conclusion, however, they acknowledge that even if they found a method for anisotropic path planning an efficient implementation is still needed.

Rowe and Richbourg (Row90b) present a new theory for the cases where homogeneous irregularly shaped regions of a plane have different traversal costs per unit distance. Their solution makes use of optics analogies, ray tracing, and Snell's Law. In the beginning they discuss established methods such as: wavefront propagation methods, improved wavefront propagation, and the Continuous Dijkstra Algorithm (CDA). As far as their algorithm is concerned they start by explaining Snell's Law, which explains the refraction or turning of the light from one region to another. The authors also define the term "well-behaved path subspaces (WBPS)" and explain some pruning methods. As far as performance goes, they analyze a worst case and an average case. They also compare their results with the ones obtained by the established methods. They conclude by saying that their method is better than the traditional ones for the following reasons: found paths are closer to the optimal; it solves conceptually simple problems quickly, sometimes dramatically so, avoiding obvious useless search directions; it avoids unreasonable jagged paths due to digital approximation; it reduces round off errors; and it avoids directional bias in paths.

Rowe and Lewis (Row89) have developed an algorithm for path-planning in three-dimensional space with respect to the following issues: energy costs proportional to path length, turn costs where paths change trajectory abruptly, and "safety costs" for the danger associated with traversing a particular path due to visibility or invisibility from a fixed set of observers. They consider free space as a set of irregular polyhedra and they

assume that energy expenditure is proportional to path length. They also consider that visibility is binary. They prove that although calculus of variations is mostly used to solve optimal-path problems, it is not the preferred method in this case. Therefore, Snell's method will be used. They prove that this method applies only on the region boundaries for their problem. By using Lagrange multipliers they developed the following generalization of Snell's Law:

$$m_1 \sin(q_1) + \left(\frac{K}{l_1}\right)\cos(q_1)\cos\left(\frac{|q_1 - q_2|}{2}\right)\mathrm{sgn}(q_2 - q_1) =$$

$$m_2 \sin(q_2) + \left(\frac{K}{l_2}\right)\cos(q_2)\cos\left(\frac{|q_2 - q_1|}{2}\right)\mathrm{sgn}(q_2 - q_1)$$

Equation 2.1

In the above formula μ represents cost per unit distance, θ represents angles for entering and leaving an area with respect to the normal to the boundary between the two regions, K is the constant of proportionality between traversal cost and turn cost, and 'l' represents the lengths of paths in each region. After they apply the A^* search method, the authors discuss the implementation of their program. They conclude by saying that this algorithm is not complete and that there are some problems with numerical errors.

CHAPTER 3
THE OFF-LINE PATH PLANNER

The autonomous navigation system is implemented on a series of PC104 computers, where each computer is assigned a specific set of tasks. More information on the architecture is presented in Chapter 5. Therefore, the path planning function is installed on a specific computer and it handles both the tasks of goal-to-goal path planning and survey path planning. The goal-to-goal software development was implemented by another student, Arfath Pasha, and as such will not be discussed further here. The survey planning software accounts for two-dimensional (flat terrain) and three-dimensional (terrain with elevation grid) path planning.

As previously stated in Chapter 1, the following information is assumed to be known:

1. The latitude and longitude coordinates of the vertices of a polygon that represents the region to be surveyed.
2. An ordered set of data for the terrain, that contains information about latitude, longitude, and altitude for points within the polygon to be surveyed. The terrain data will be stored in an evenly spaced grid.
3. The distance between adjacent surveying lines for the planned path, i.e. the resolution of the survey.
4. The locations of obstacles located within the area to be surveyed. These obstacles are defined by a polygon whose latitude and longitude vertice coordinates are specified.

5. The maximum allowable pitch angle and roll angle for the vehicle.

6. The length and width of the vehicle performing the survey.

7. The minimum allowable radius of curvature for the vehicle.

The objective is to determine an efficient survey path that would sweep the given field without violating vehicle constraints. To obtain this objective, a series of steps were taken. First, successive survey paths are generated based on varying the principal direction axis of the survey from zero to 180 degrees. Second, each path is attributed a cost based on the pitch and roll of the vehicle at each location along the path as well as the length of the path. Third, the minimum cost path is chosen and all its defining points are written to a file. Lastly, if the minimum path contains any pitches or rolls that violate the vehicle constraints, those elements are recorded as obstacles and an obstacle avoidance algorithm is applied to insert a sub-plan around this region. As a result of the above steps, an effective safe path is obtained for the given terrain and vehicle characteristics.

Terrain Data

Obtaining three-dimensional terrain data can be accomplished in two ways, i.e. using real data or creating artificial test data. Real data can be obtained by using government data, which is available for public use, or, by recording data from a field with the use of a special vehicle. Artificial test data can be generated for a field using a special algorithm or, by generating a field where the heights have been created randomly.

The test input data that was used for development and testing purposes is stored in a 32×32 matrix. While the concept of random generation is self explanatory, the issue of government data and how to use it is more complex.

In an effort to represent earth science data in a better way, the United States Geological Survey (USGS) has chosen the Spatial Data Transfer Standard (SDTS) as their new standard for earth related data and made it mandatory for all federal agencies. Since the SDTS is a relatively new format, all programs and algorithms are connected to the previous standard, the Digital Elevation Model (DEM). SDTS is a form of encoding for earth-referenced spatial data and is designed to work with vector and raster data structures. It is available for the entire United States and for Alaska, by state or by quadrangle name. DEM is a collection of elevation points for a specific area and it represents the old way of storing earth related data. There are four types of data based on the number of data points contained: 1:24,000-Scale, 1:100,000-Scale, 1:250,000-Scale, and National Elevation Data Set. For our purpose the 1:24,000-Scale DEM (SDTS) was chosen because of its accuracy and for its large use in different applications. The only drawback is the fact that there is a 30m distance between data points but it has the best accuracy out of all formats. Since most applications use DEM, there is a way of converting SDTS to DEM. The process starts by unzipping the SDTS, which comes as a gzip-compressed tar file. This step produces numerous files with the ‘ddf’ extension. The only one needed has the following format:

XXXXCEL0.DDF

In the above format XXXX represents any combination of letters and digits based on the specific cell that is needed. The next step is to use the program SDTS2DEM, which can be obtained from www.geopotential.com, to convert the SDTS to DEM. The result is a file that has the DEM format and that can be used with most applications.

The details of the SDTS, DEM, and SDTS2DEM are complex and it is not the intent of this paper to discuss them. However, Appendix A contains numerous web addresses that provide more information. Also Appendix B contains the format for a DEM.

There are many applications that use 1:24,000-Scale DEM's as inputs and display them either in 2D or 3D. For example, Terragen (Figure 3.1) displays DEM's in 3D as photographs.

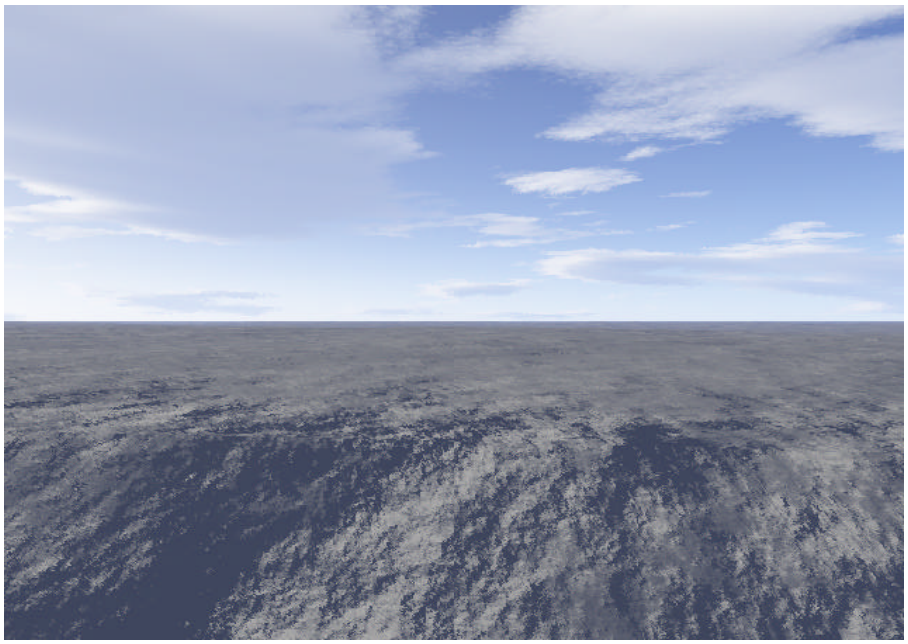

Figure 3.1. Display of DEM data for Gainesville, Florida, using the Terragen application.

Others, while taking DEM's as inputs, can display them as 3D image like Leveller, DEM Tools, 3DEM, dlgv32, Microdem, Wilbur, and Worldgen. All the ones listed previously are designed also to convert between different types of file formats such as DEM and DXF (AutoCAD file). Appendix A contains web addresses for more information on these applications. The display of the data is not the primary interest of this research. However, a computer program using the Open Inventor graphics library

functions was written in order to display the data. One data format that can be imported into Open Inventor is the Virtual Reality Modeling Language 97 format and for this reason the DEM format data is converted to this format. The only problem that arises here is that the version of Open Inventor that supports VRML 97 was not available at the time of implementation. However, efforts were under way to implement a new version under RT Linux, which supports such a file format. Figures 3.2 through 3.4 show pictures of data converted from DEM to VRML97 by using two different programs.

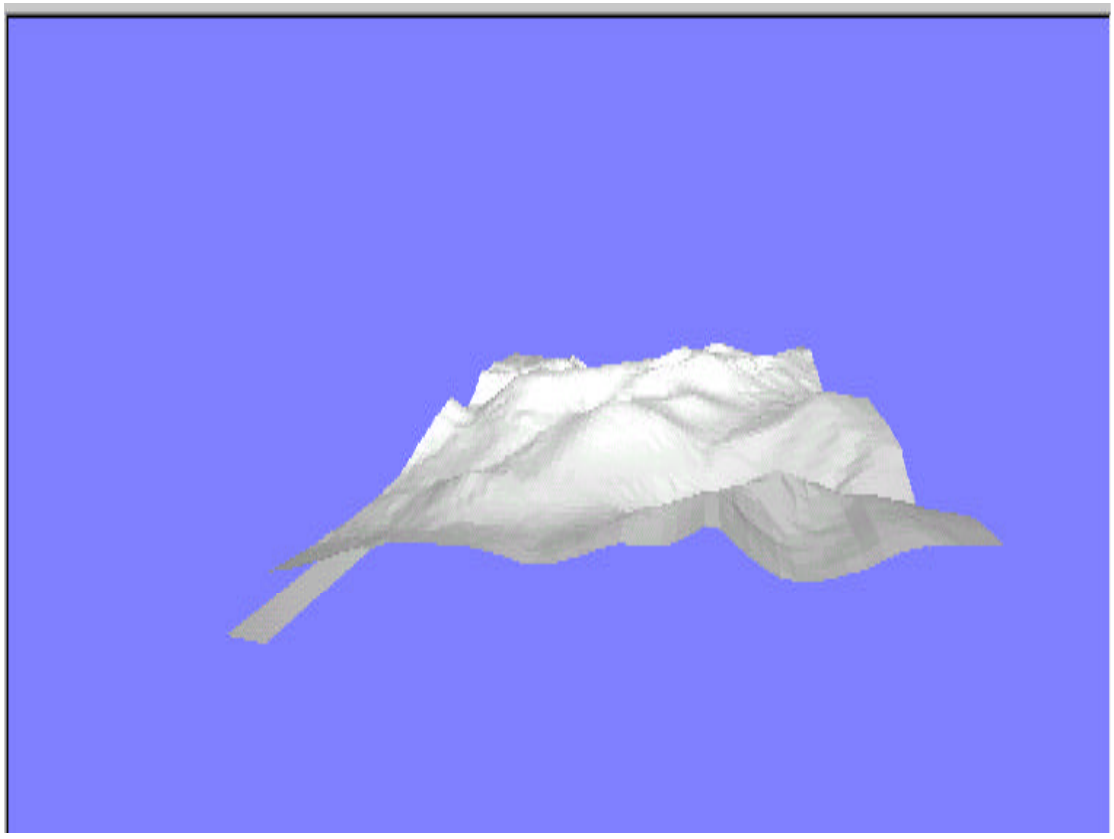

Figure 3.2. Display of terrain data for Gainesville, Florida, using the Leveller application

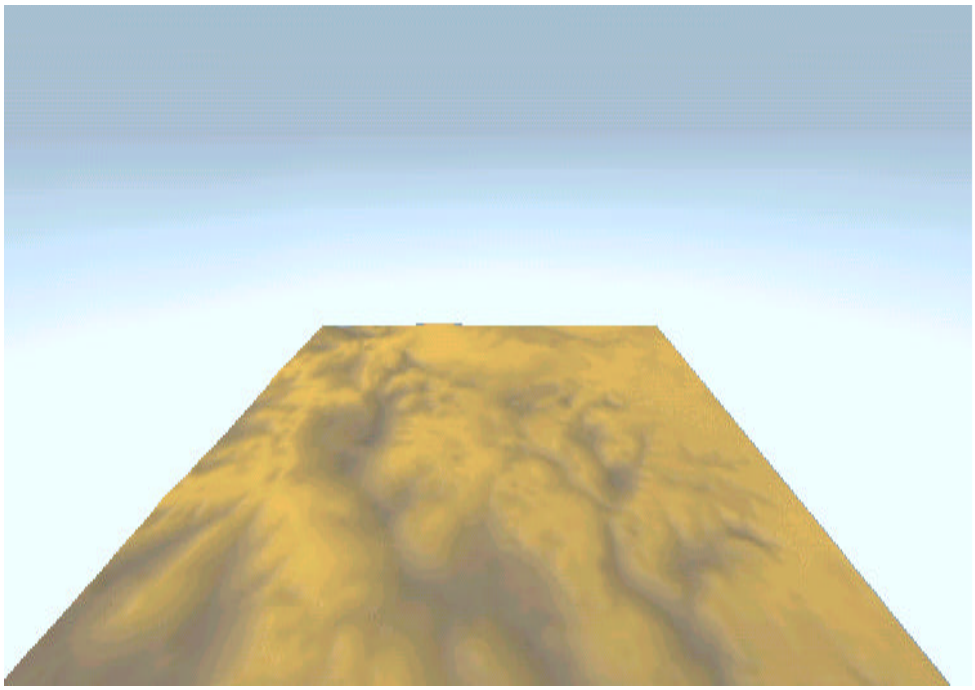

Figure 3.3. Display of DEM data for Gainesville, Florida, using the 3DEM application

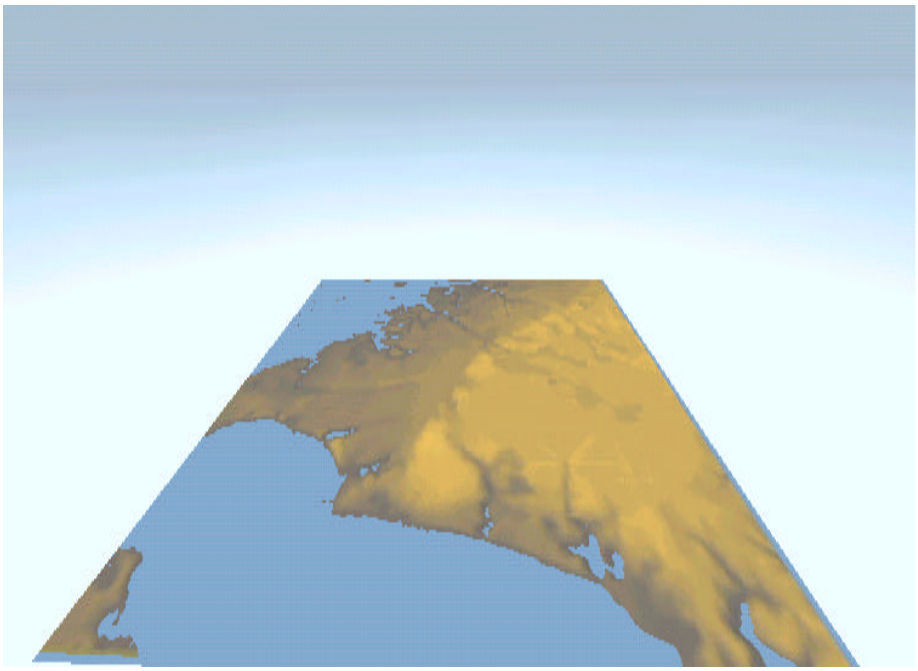

Figure 3.4. Display of DEM data for Panama Beach, Florida, using the 3DEM application

Another method of using the DEM's is by using the same converters described above but this time the needed files are in x, y, z format. The converted files, however, have around 250,000 data points, which is too big for an algorithm to analyze. Therefore, a special program has been written so that only every other 32^{nd} data point is written to a specific file. Finally, this file has the data necessary to fill a 32×32 matrix. It is easy now to use this format either as input for Matlab or Open Inventor. Figure 3.5 shows a random generated field rendered using Matlab.

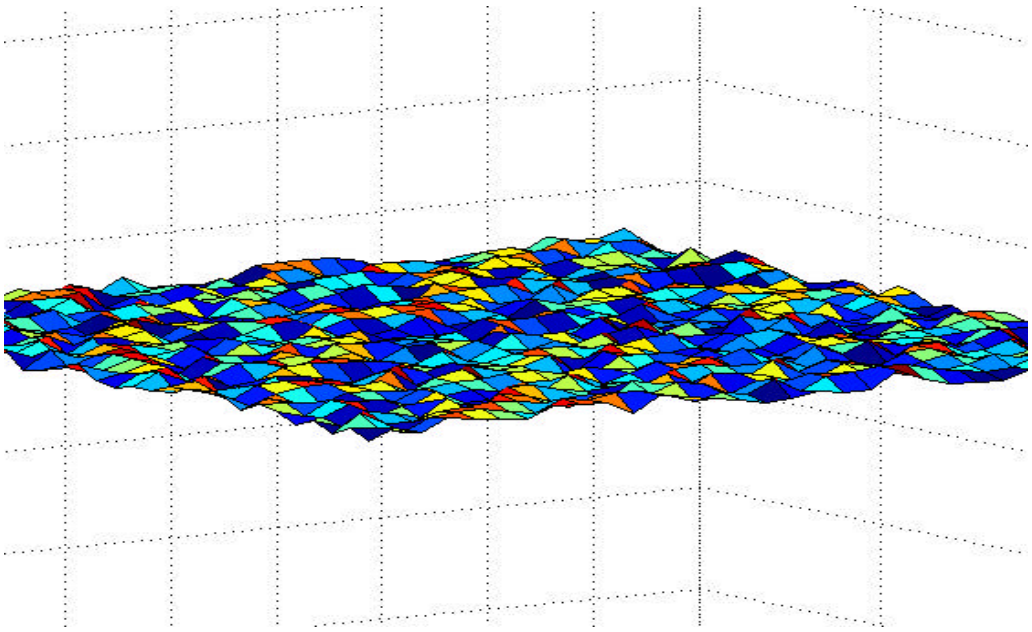

Figure 3.5. Random generated terrain

Two-Dimensional Path Planner

Rankin (Ran93) has initially developed the two-dimensional survey path planner. The author has modified the software in order to make it modular and in accordance with

the new architecture. By making it modular, the algorithm is broken into functions that handle specific tasks.

The first step in the algorithm is to convert the given data from latitude and longitude coordinates to a North-East-Down (NED) coordinate system, and then to a local x, y, z coordinate system. The formulas for converting between these coordinate systems are shown in Equations 3.1, 3.2, 3.3, and 3.4.

$$north = (latitude - zeroLatitude) \times d2r \times earthRadius$$

Equation 3.1

$$east = (longitude - zeroLongitude) \times d2r \times earthRadius \times \cos(zeroLatitude \times d2r)$$

Equation 3.2

$$x = north \times \cos(theta) + east \times \sin(theta) - zeroNorth \times \cos(theta) + zeroEast \times \sin(theta)$$

Equation 3.3

$$y = -north \times \sin(theta) + east \times \cos(theta) - (-zeroNorth \times \sin(theta) + zeroEast \times \cos(theta))$$

Equation 3.4

In the above equations, ‘zeroLatitude’, ‘zeroLongitude’, ‘zeroNorth’, and ‘zeroEast’ represent the coordinates of the origin, ‘d2r’ represents the conversion factor from degrees to radians, and ‘theta’ represents the angle of surveying measured relative to the x axis.

The next step is to find the intersections between lines parallel to the x-axis and the lines formed by the boundary points of the polygon to be surveyed. The user enters the distance between the lines. The lines represent the rows that the vehicle will follow in order to survey the field. Finding the intersections is handled by a specific function that

checks the intersection of two line segments based on the formula for a line, which is shown in Equation 3.5:

$$y = a * x + b$$

Equation 3.5

The boundary line segments are defined by a structure that contains information about x and y coordinates of start and end points. The lines parallel with the x-axis are defined by a structure that contains information about the 'a' and 'b' in Equation 3.5.

In order to make the algorithm more efficient, the first row is found by going down on the y-axis until there are no more intersections between the rows and the boundary line segments. The intersection points are added in a linked list regardless of their order in the path. If the resulted number of rows is odd then the center row is added twice. In this way there is always an even number of rows. This accounts for the fact that if there is an odd number of rows the middle row will be traveled twice: once as the second row in decreasing x order, and the second time as the next to last row in an increasing x order.

The next step is to put the points in order. This is done in a specific function that puts the points in order based on the row order and on the direction of travel. At this point in the algorithm it is necessary to define the possible directions of travel: increasing, when the x-coordinate is increasing, and decreasing, when the x-coordinate is decreasing. A $160m \times 160m$ field that is surveyed at 5m distances between the rows is shown in Figure 3.6.

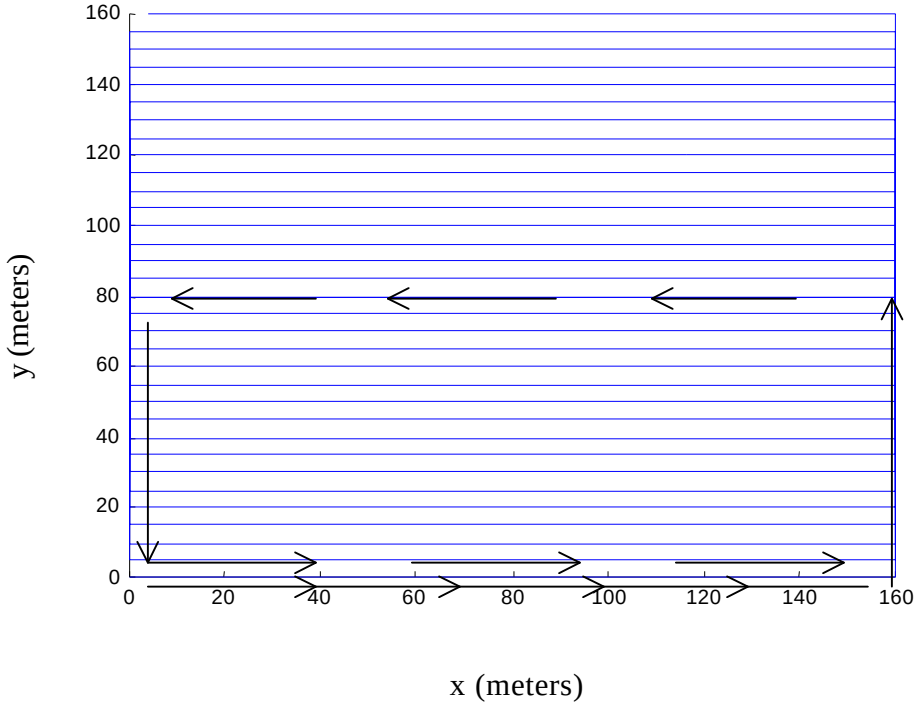

Figure 3.6. 2D Path planning at 0^o and 5m distance between lines

The next step is to account for different surveying angles. The three-dimnesional algorithm will eventually calculate the cost for a path at a given survey angle. Therefore, the capability of surveying at different angles is needed. Figure 3.7 shows a survey path created with a survey angle of 80^o. In the case of angled path planning, the starting point will move from the corner of the grid to a different point based on an effective way of surveying a given field. For an example, for an $160m \times 160m$ field, the starting point has moved from 0, 0 to 140, 0 as shown in Figure 3.8.

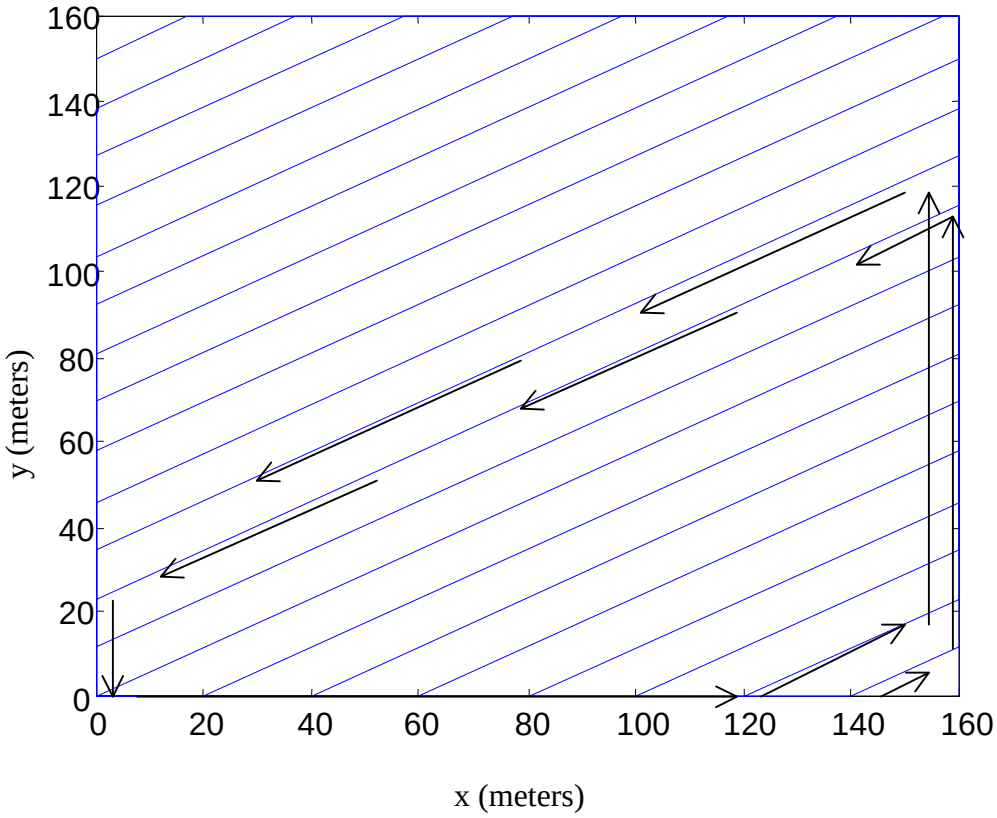

Figure 3.7. 2D Path planning at 30^{o} and 10m between lines.

The last step for the 2D path planning algorithm is to account for the turning radius of the vehicle. This task is accomplished by a specialized function created by Rankin (Ran93). The results of this function as well as the results of the entire algorithm are shown in Figure 3.8.

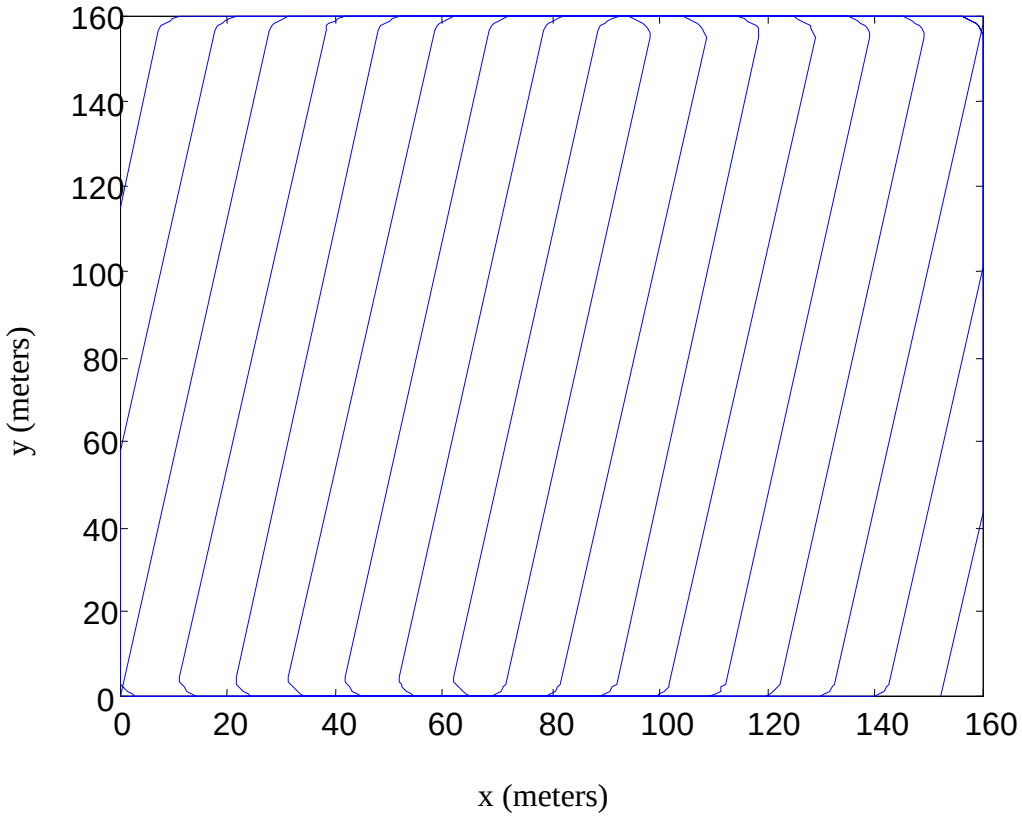

Figure 3.8. 2D Path Planning at 80^o for a turning radius of 5m.

Optimal Path

The goal at this point is to find the optimal path for surveying the field. In order to do this task, the user will specify a survey angle increment. In other words, the algorithm will compute paths from 0^o to 180^o in specified angle intervals, as shown in Figure 3.9. The points in each path will not be recorded, but rather a cost associated with each of them will. The cost is based on pitch, roll, and the length of the path. Cost associated with a point is based on the maximum allowable rotation for a specific vehicle. For the Navigation Test Vehicle (NTV), the specific characteristics were that it could not travel on slopes larger than 15^o. Therefore, if the pitch or roll were greater than 15^o, the

point had a very high cost associated with it, in this case 9999. Otherwise, each point had a cost equal to the distance of the point from the previous point.

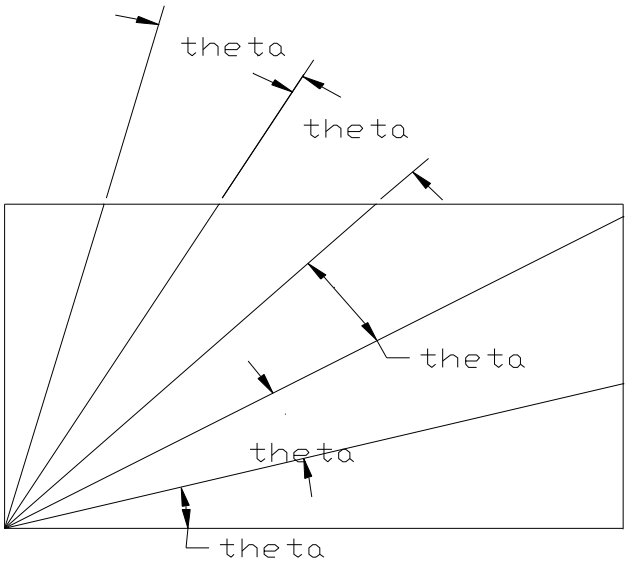

Figure 3.9. Path planning at θ angle intervals

The pitch is calculated by a specific function that determines the angle between the current point and a point situated at a specified distance at 90^o relative to the survey angle in front of the current point. The roll is calculated by the same function between the current point and the point situated at the same distance but at 0^o relative to the survey angle from the current position. Although the distance is variable, for this case it has been chosen to be 0.5m. In order to calculate the slope between two points it is necessary to know the x, y, and z values for those points. The x and y coordinates are entered as initial conditions to the function but the height at each point needs to be computed. This task is handled by a function that calculates the height at any point in a square knowing the heights at the four points (Seg84). The formulas used for this function are shown in

Equations 3.7, 3.8, 3.9, 3.10, and 3.11. The four points are the corners of the grid cell in which the specified point is located as shown in Figure 3.10.

$$c1 = field[i][j]$$

Equation 3.7

$$c2 = \frac{(field[i+1][j] - field[i][j]}{2 \times length}$$

Equation 3.8

$$c3 = \frac{(field[i][j+1] - field[i][j])}{2 \times length}$$

Equation 3.9

$$c4 = \frac{(field[i][j] - field[i+1][j] + field[i+1][j+1] - field[i][j+1])}{4 \times length^2}$$

Equation 3.10

$$height = c1 + c2 \times s + c3 \times t + c4 \times s \times t$$

Equation 3.11

In the equations above, c1, c2, c3, and c4 represent constants, ‘field’ is the name of the matrix, ‘length’ is the distance between data points, ‘height’ is the needed height, ‘s’ and ‘t’ are the axis of the local coordinate system, whose origin is at the lower-left corner. Since finding the grid cell is not an obvious fact, it is handled by another function. It loops through the corners of each grid cell, both on x-axis and y-axis, until the right cell is found.

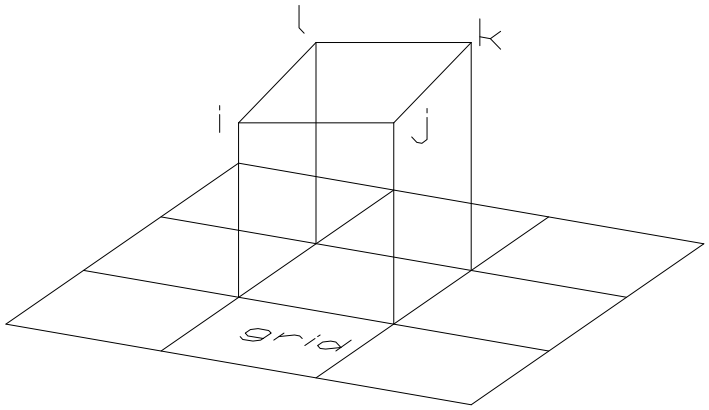

Figure 3.10. Finding a height in a cell defined by i,j,k, and l.

Once the cost for each path has been computed, the minimum value is selected and the path associated with it is recomputed. This time however, all the points that make up the path are recorded to a special file. In this way, memory management is not as issue.

Obstacle Avoidance

The obtained minimum path may still contain inaccessible points and, therefore, an obstacle avoidance algorithm will be performed at these locations. Before that, however, some of these points might be repeated, especially in the case where the original number of rows is odd. In that case obstacles on the middle row will be added twice, once in the beginning of the file and once at the end. If this case occurs, the additional obstacles are erased.

The algorithm, however, was designed to work with polygons and not points. Therefore, a virtual polygon will be created around each of these inaccessible points. The shape of the polygons is square and the size of the polygons is variable. The algorithm starts with the length of one side of the square being equal to 1m. After applying these dimensions the algorithm checks to see if the slopes created by the corners of the polygons are favorable for the vehicle's movement. If it's not, then the size of the polygons is increased until travel is allowed in the specified direction. Once the size has been decided, a special linked list is created and filled with the coordinates of these virtual polygons. Then, the minimum path is checked for intersection with the known obstacles, whose positions are recorded by the user in a special designated file. This file has a specific format that is shown in Appendix C. Results are shown in Figure 3.11.

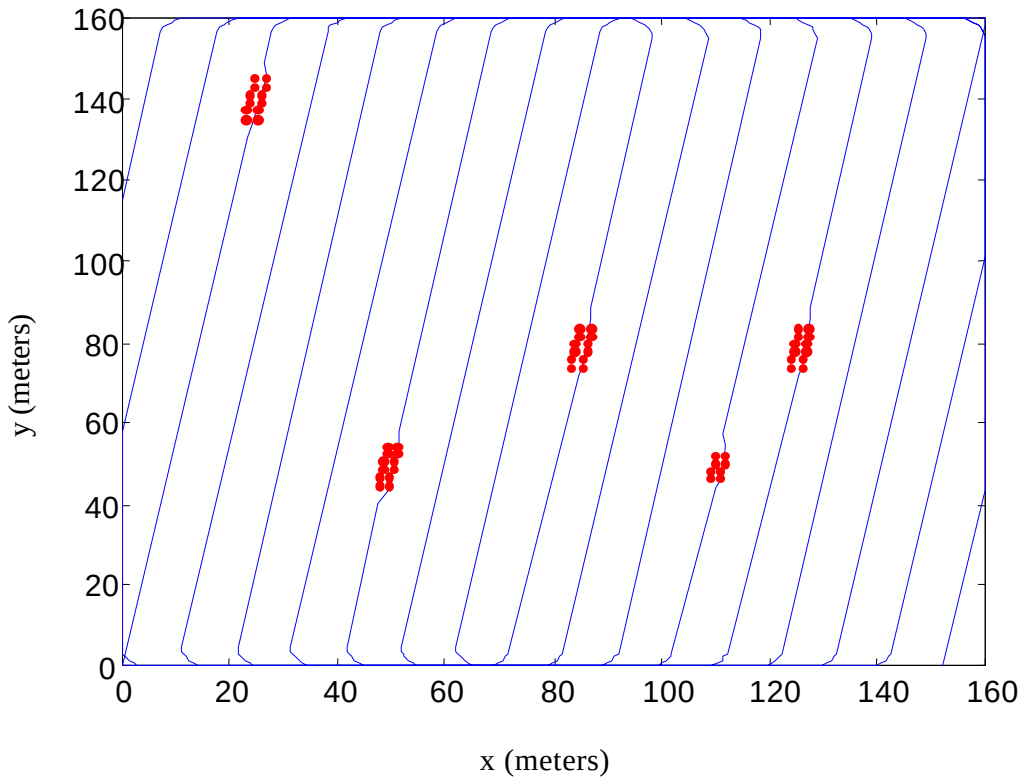

Figure 3.11. Obstacle Avoidance for Non-intersecting Obstacles

In order to avoid the collision between the vehicle and the obstacle, an expansion algorithm (Ran93) is performed on each obstacle and then the intersection of the path and the expanded position is checked. Also, if two or more obstacles are intersecting with each other, a special function will combine them into one obstacle simplifying in this way the obstacle avoidance algorithm. Results are shown in Figure 3.12.

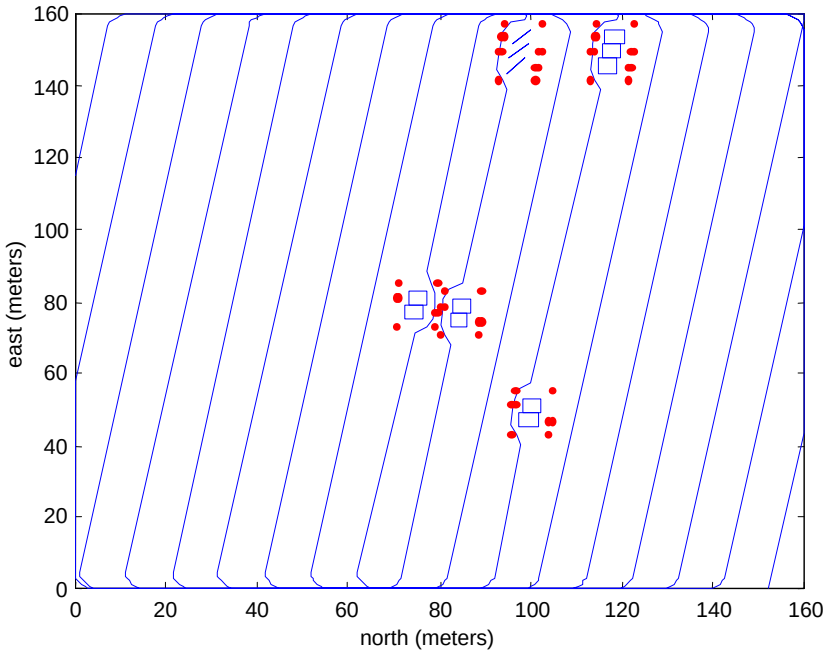

Figure 3.12. Obstacle Avoidance with Intersecting Obstacles

The obstacle avoidance task is handled by a specialized function that is designed to work for either known obstacles, such as trees or houses, as well as the obstacles that came from un-favorable slopes. Since it is a complicated mission this function has its own algorithm.

Before the actual obstacle avoidance algorithm starts, the coordinates for the obstacles are read from the specialized file. This applies for either the ones that are written during the finding of the bad slopes process or the ones that are introduced by the user as manmade obstacles, such as a house or trees. Then the vehicle's characteristics such as length and width are read from a different file, which has to also be created by the user. This file has also a specific format, which is shown in Appendix C. Getting the right dimensions for the vehicle is very important because very bad results can be obtained if the numbers do not match reality. Then the obstacles are checked for intersection between each other and the expansion algorithm (Ran93) is applied to the resulted obstacles.

The first step is to determine the slopes of the lines that may intersect the obstacles. Although there might be intersections between the obstacles and the side line segments, which connect the ends of the rows, these will be ignored. Therefore, considering Equation 3.5 the variables 'a' and 'b' need to be defined for each row. Since checking the intersections is done in the local coordinate (x, y, z) system the 'a' is always zero. That leaves only the 'b' to be determined. This was done by taking a line defined by Equation 3.5 and by bounding it by the pair (x1, y1) at one end and by the pair (x2, y2) at the other end. This procedure leaves two equations with two unknowns shown in Equations 3.12 and 3.13.

$$y_1 = m \times x_1 + b$$

Equation 3.12

$$y_2 = m \times x_2 + b$$

Equation 3.13

The result is shown in Equation 3.14:

$$b = \frac{y_1 \times x_2 - y_2 \times x_1}{x_2 - x_1}$$

Equation 3.14

In order to find all the intersections between the obstacles and the surveying lines an outside loop goes through the list of obstacles while an inner loop goes through the row numbers.

The next step is to arrange the intersections in the right order based on the direction of travel and based on the order of the obstacle on a specific row. In order to effectively arrange all the obstacles in the right sequence, every time the intersection between each obstacle and a specific line is checked, the search begins at the start of that line. Since the most common case for the obstacles is that their shape is rectangular, there are most commonly two intersections between the obstacle and a line. The algorithm goes through the list of points on one line until the x-coordinate of the first intersection is in the right place. There are three different cases, increasing x-coordinate, decreasing x-coordinate, and increasing-decreasing x-coordinate. The most complicated case is the increasing-decreasing, which arises when there is an odd number of rows and, therefore, the middle row will be traveled twice. The row is traveled the first time in the beginning of the algorithm in increasing order and then it is traveled a second time at the end of the algorithm in decreasing order. Since the algorithm starts searching in the beginning of the line this case is handled in the same way as the other two cases. Also the algorithm checks for the order of the intersections for each obstacle based only on the direction of travel. If the x-coordinates of the intersections are not in order they will be swapped.

The next step is adding the intersection points and the boundary points of the obstacle. The concept is simple: a loop goes through all the intersection points. Since they are already in order as soon as the first intersection of one obstacle is found, the length between the two intersection points is checked. Since there is a polygon there are two lengths between the points. One is by traveling clockwise and the other by traveling counter-clockwise as shown in Figure 3.13.

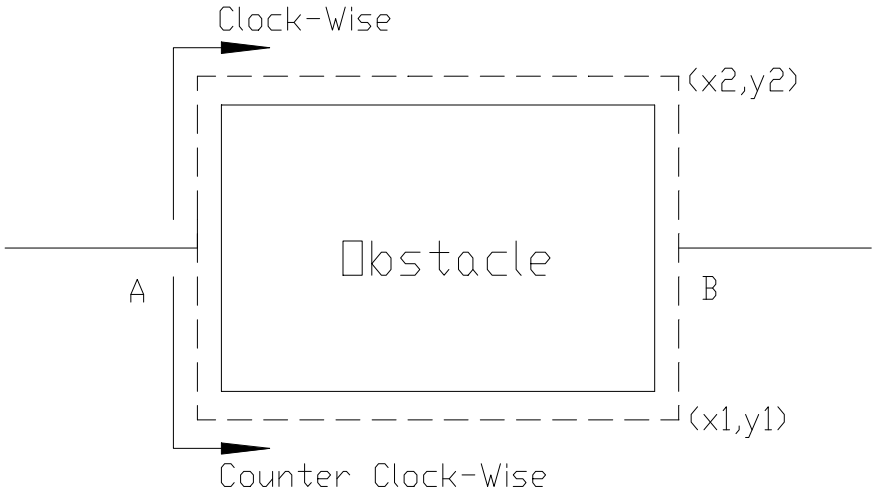

Figure 3.13. Possible directions of travel around an obstacle.

Since the algorithm minimizes traveling time, the shorter of the two distances is needed. A special function handles the task of calculating which direction should be followed. If the clockwise length between the points is shortest then all the points in between the intersection points and to the left of the first point will be added, otherwise the points to the right of the first point will be added. In the case that the clockwise length and the counter-clockwise length are equal, the clockwise direction will be followed. The function sets as the start the line segment on which the first intersection point is situated

and as the end the line segment on which the second point is situated. Since in the beginning of the avoidance algorithm all the boundary points of each obstacle have been assigned to line segments, the direction deciding procedure adds the different sides of the line segment pending on the direction. For example, if one endpoint of a line segment is described by the pair (x1, y1) and the other endpoint is described by the pair (x2, y2) and the direction is clockwise, then by going clockwise the first encountered endpoint has the coordinates (x1, y1). Therefore, all the points described by the (x1, y1) coordinates and situated by increasing line segments are being considered as shown in Figure 3.13. The distance between every two points is calculated using the Pythagorean formula, which is shown in Equation 3.15, and all the resulted distances are added together.

$$length = \sqrt{(y_1 - y_2)^2 + (x_1 - x_2)^2}$$

Equation 3.15

The same procedure is true for the counter-clockwise direction. At the end of the function the two lengths are compared and the function returns a true value if the clockwise direction is shorter than the counter-clockwise. If one of the directions requires travel outside the boundaries of the entire field, then the algorithm automatically selects the other direction as valid.

Once the direction of travel between the intersection points is established the algorithm starts adding the points based on the same line segment approach as presented above. The line segments are organized in clockwise increasing order by the expansion procedure.

Special cases are shown in Figures 3.14 through 3.18. For example, in 3.14 and 3.15 a single obstacle is shown on the middle row before and after smoothing.

Smoothing deals with accounting for the turning radius of the vehicle that is surveying the field. As it can be seen in each pair of figures, in the first picture every corner is at 90^{o}, while in the second figure the corner are rounded to allow the vehicle to follow the path. In 3.16 and 3.17 one obstacle is shown on the middle row but using a 20^{o} survey path planning angle. In Figures 3.18 and 3.19 there are multiple obstacles on the sides and center.

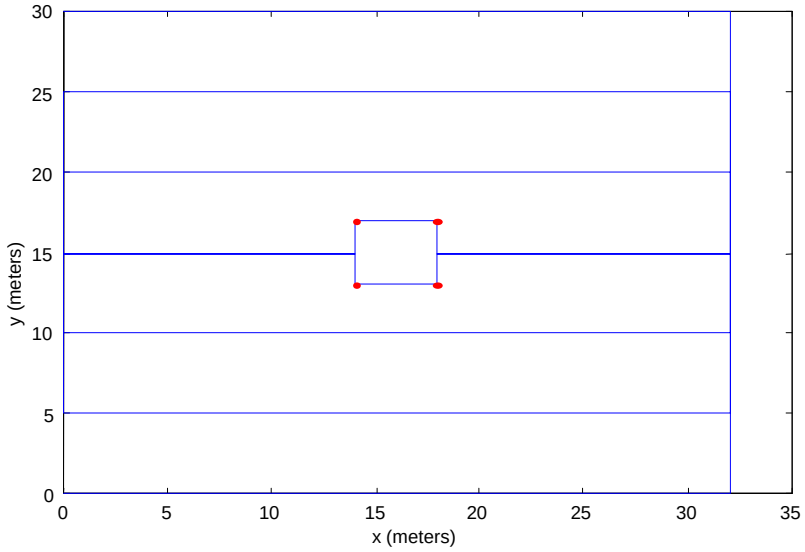

Figure 3.14. One obstacle on the middle row before smoothing.

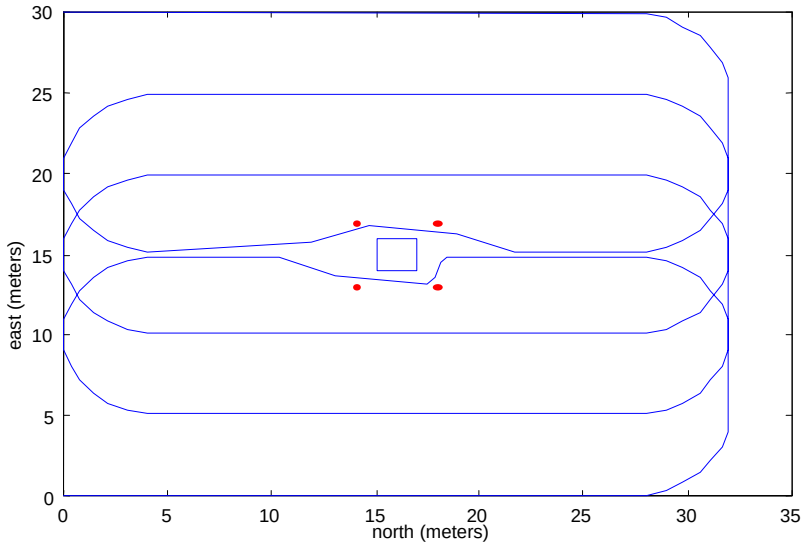

Figure 3.15. One obstacle on the middle row after smoothing

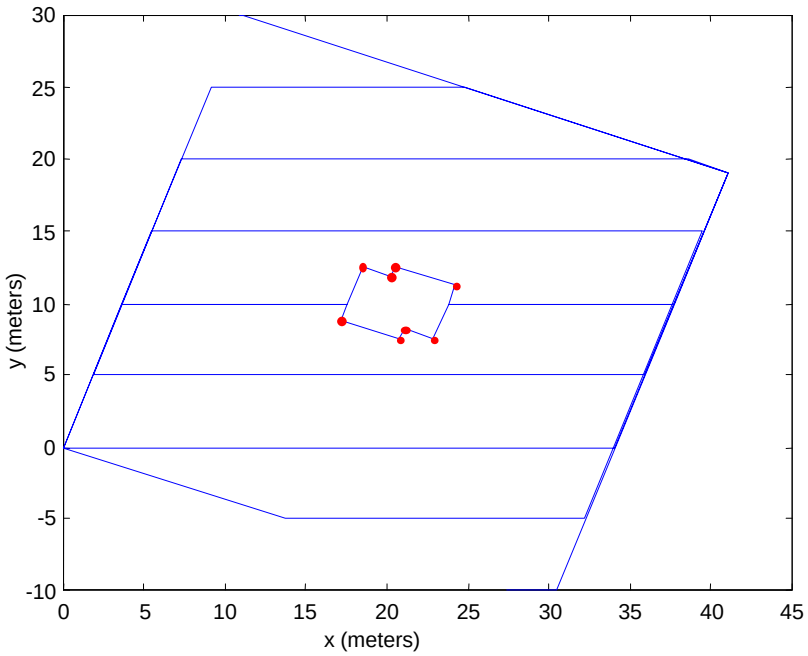

Figure 3.16. Same case but on a rotated coordinate system.

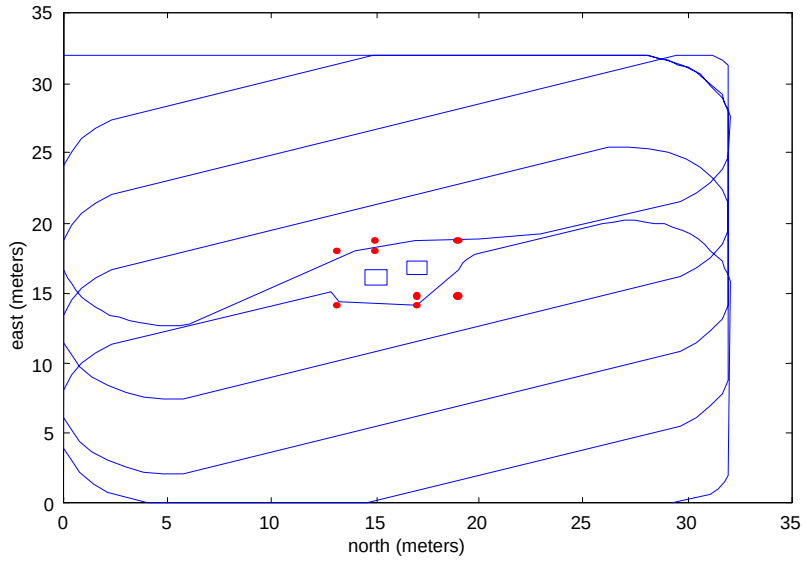

Figure 3.17. Same case after smoothing occurred.

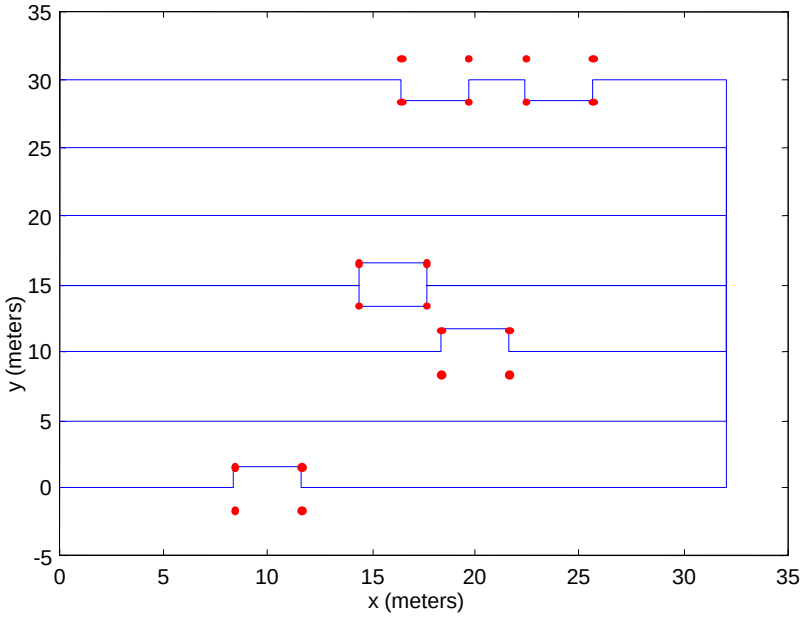

Figure 3.18. Multiple obstacles on the sides and center

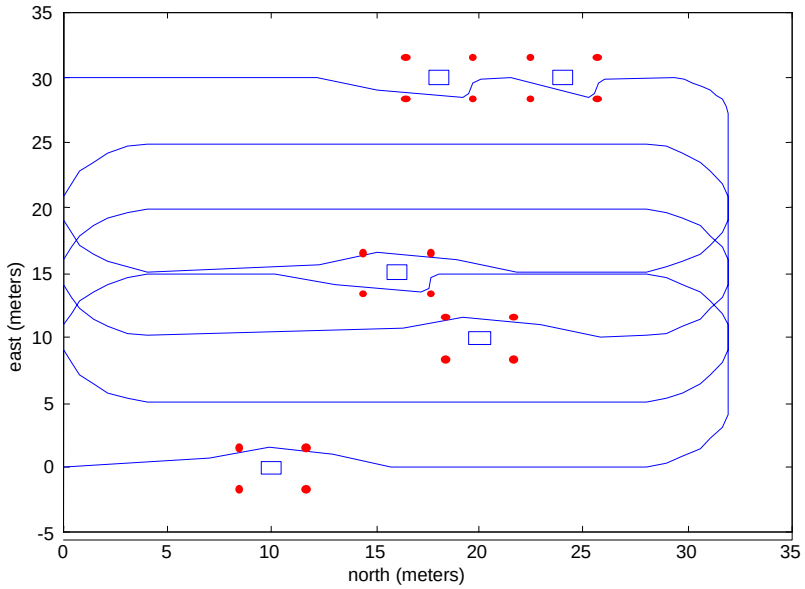

Figure 3.19. Same field after smoothing

CHAPTER 4
RESULTS / CONCLUSION

The two types of data discussed in Chapter 3 are used for testing the algorithm. However, they present some shortcomings as far as real testing goes. While the random generated terrain often either creates too many slopes or no slopes, the government data has too many gaps in between grid points. As stated before, there is a distance of 30m between any two points on a Digital Elevation Model (DEM). The data is accurate on a large scale but since the Navigation Test Vehicle's dimensions are very small in comparison with the represented terrain the data is not as useful.

Some approaches include interpolating between points. However, the best method to obtain real terrain data is to select a field and apply the sweeping method recording vehicle position and orientation data. In other words, the field is manually picked and the Navigation Test Vehicle sweeps the specified terrain recording as much data as possible. For testing purposes, a field at the Bandshell at the University of Florida was selected. The sweeping was done at one-meter intervals and the data was recorded at a two hertz rate at a vehicle velocity of 3miles/hour. The top view of the recorded data is shown in Figure 4.1. A three-dimensional side view is shown in Figure 4.2.

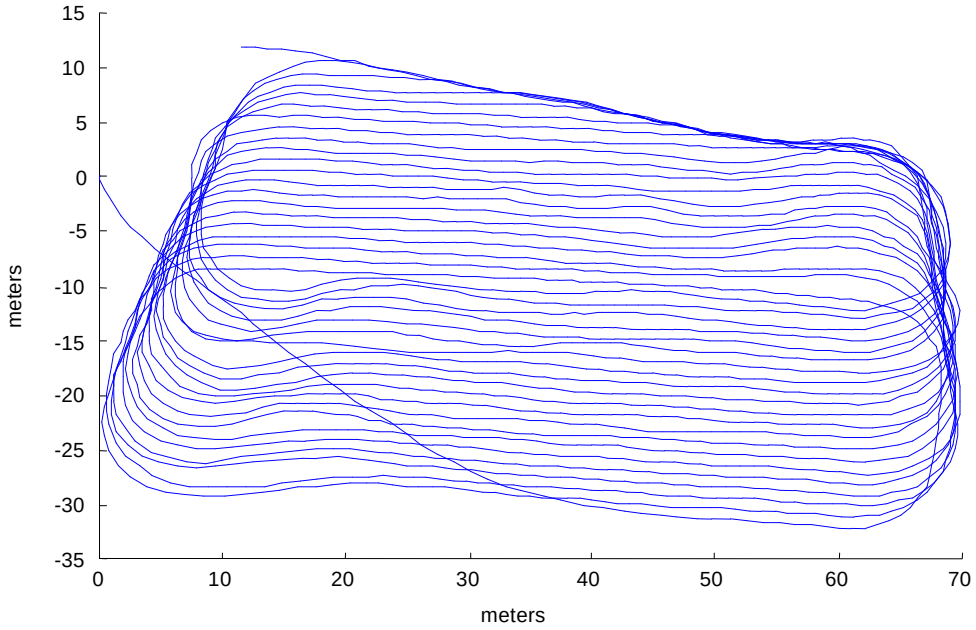

Figure 4.1. 2D Data for a field at the University of Florida Bandshell

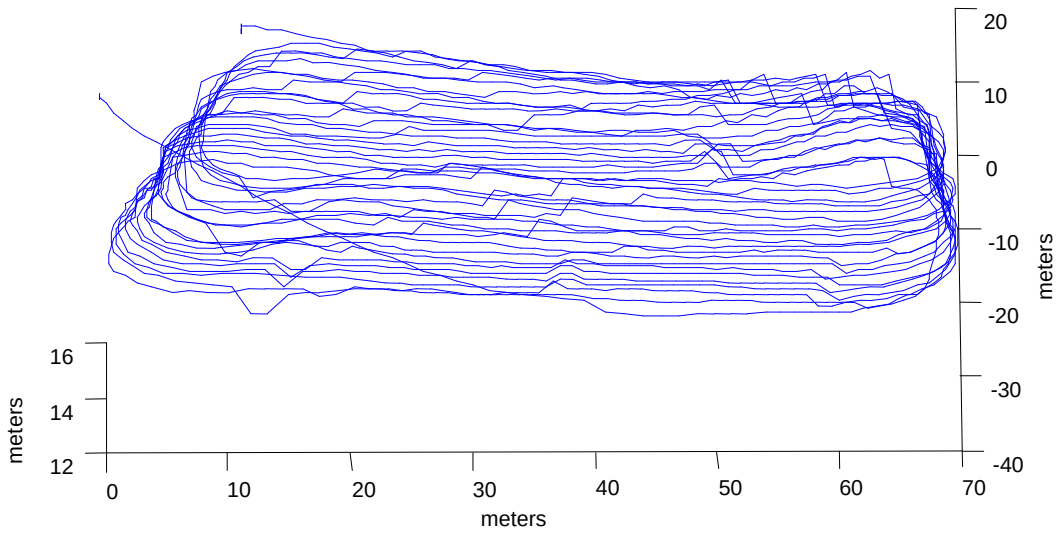

Figure 4.2. Three-dimensional view for the same field

To achieve the results shown in Figures 4.1 and 4.2 the file that was recorded during the manual sweep process was modified. The reason is that the obtained file contained information that was not needed. Therefore, only the second, third, and fourth column of the vehicle data log file were transferred to a different file. Since the original data was latitude, longitude, and altitude it was converted to the North-East-Down (NED) format before it was recorded to the new file. All these operations are handled by a specific function. Once the file is created, the function should not be used unless there is a new input file. As it can be seen in the Figures 4.1 and 4.2 the data is ordered, but the spacing of points is variable. On an average there are values for points situated every one meter but there are locations with no data. The recorded data has been arranged in a grid. However, it is organized in a grid of 70×44 size and not the 32×32 dimension that was specified in Chapter 3. The reason was to get a better accuracy for the data. In other words, the field was enclosed in a grid whose dimensions were found by finding the minimum and maximum x and y coordinates and subtracting them from each other. Handling the minimum and maximum is done in the same function as above. The number of rows and columns is found by dividing the difference between the maximum and minimum by the distance between the data points. In order to have fewer missing data points the grid was filled at every five meters. The resulted height grid is shown in Figure 4.3 and Figure 4.4.

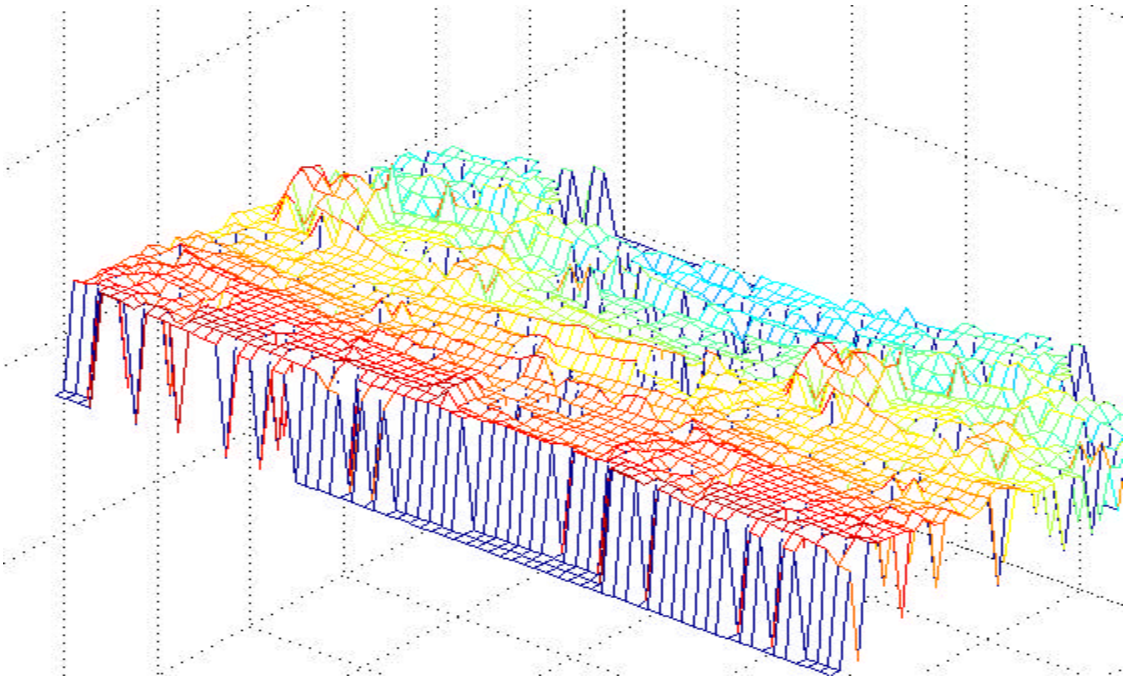

Figure 4.3. Field arranged on a 70m × 44 m grid with data at 5m spacing

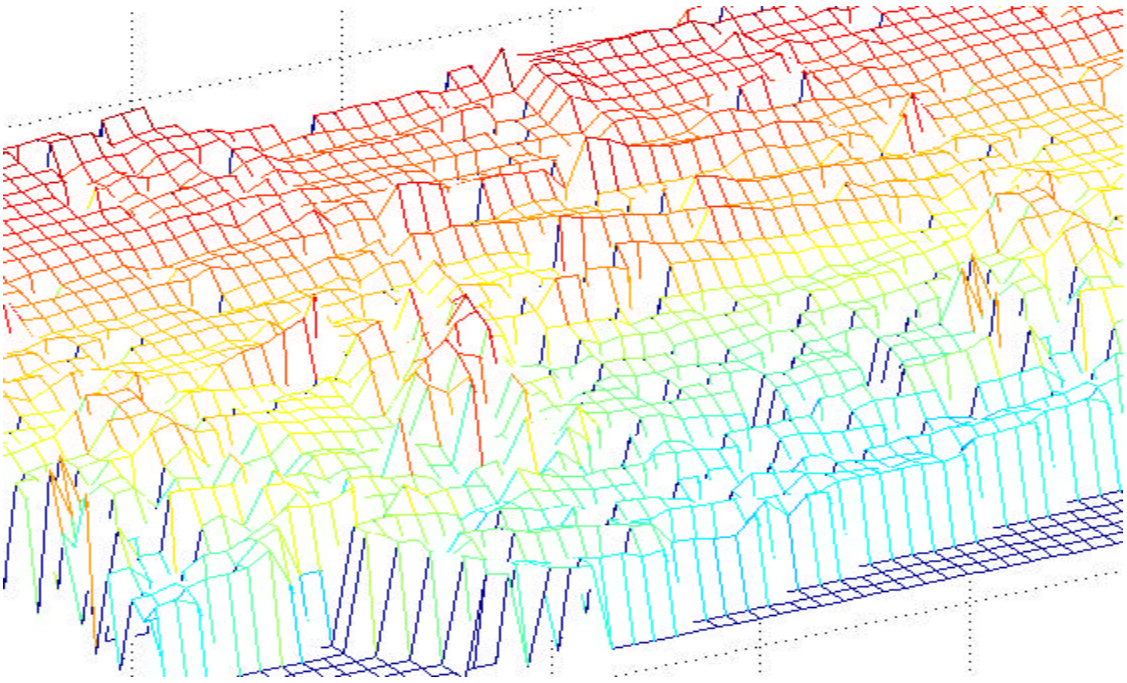

Figure 4.4. Same field as in Figure 4.3 but opposite view.

In Figures 4.4 and 4.5 the missing height values have been given a value of 12m although any value would have sufficed. For this field, the maximum height is 14.3m and lowest height is 13.54m. Even if having both these heights occurring next to each other on the map it would create a slope of 37^{o}. However, chances are that they are not occurring next to each other and also by simply observing the field there are no critical slopes of 15^{o}, which is the maximum slope value that the vehicle can traverse. Therefore, for the purpose of testing the efficiency of the program, the critical value was lowered to 5^{o}.

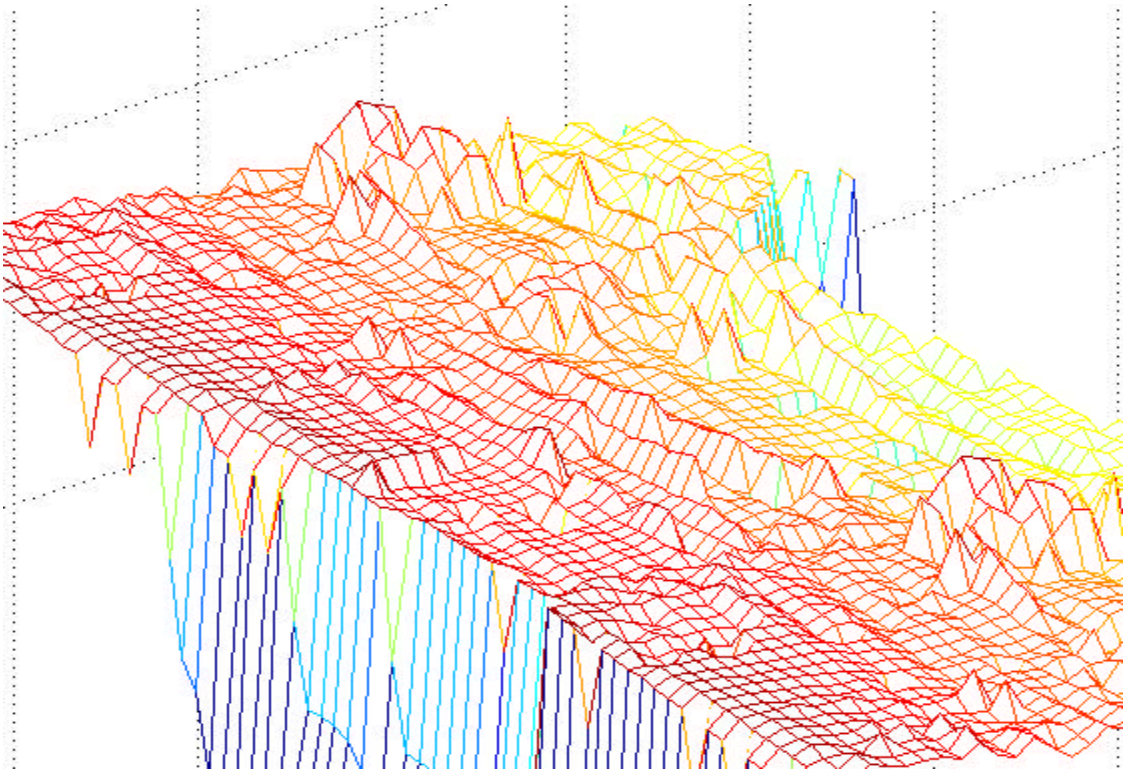

Figure 4.5. Same grid after missing heights have been added

Figures 4.6 and 4.7 show the path that has been computed for a section of the field shown in the above figures.

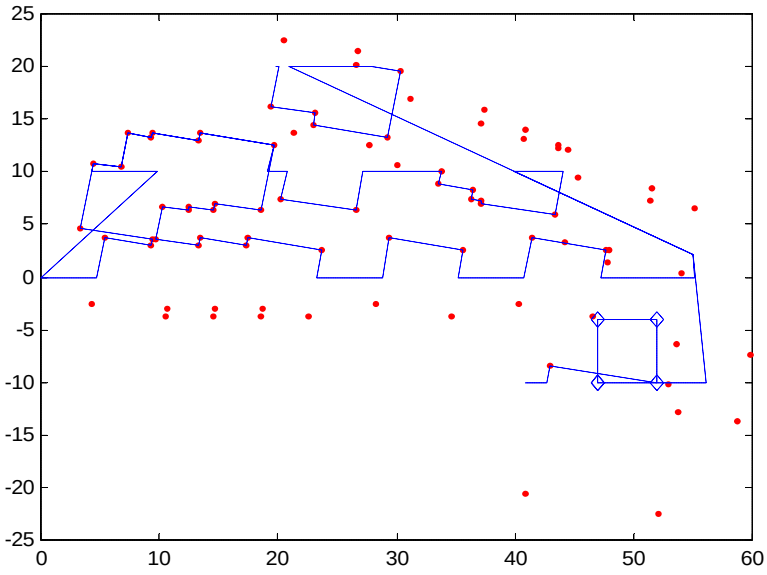

Figure 4.6 Path shown in 2D for a part of the field

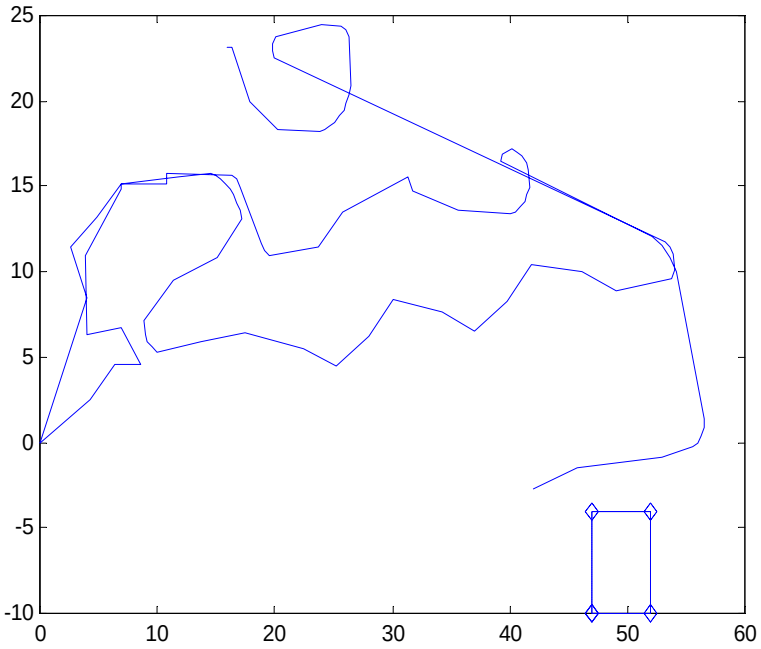

Figure 4.7 Same path as in Figure 4.6 after smoothing has been applied

In Figures 4.8 and 4.9 the path is shown on the field.

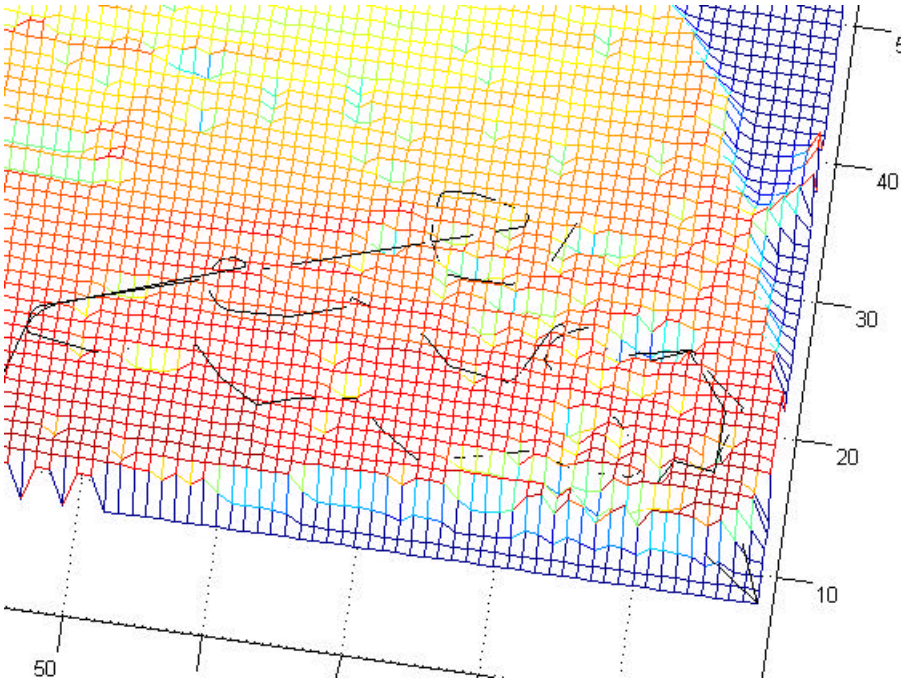

Figure 4.8 Path is shown on the field

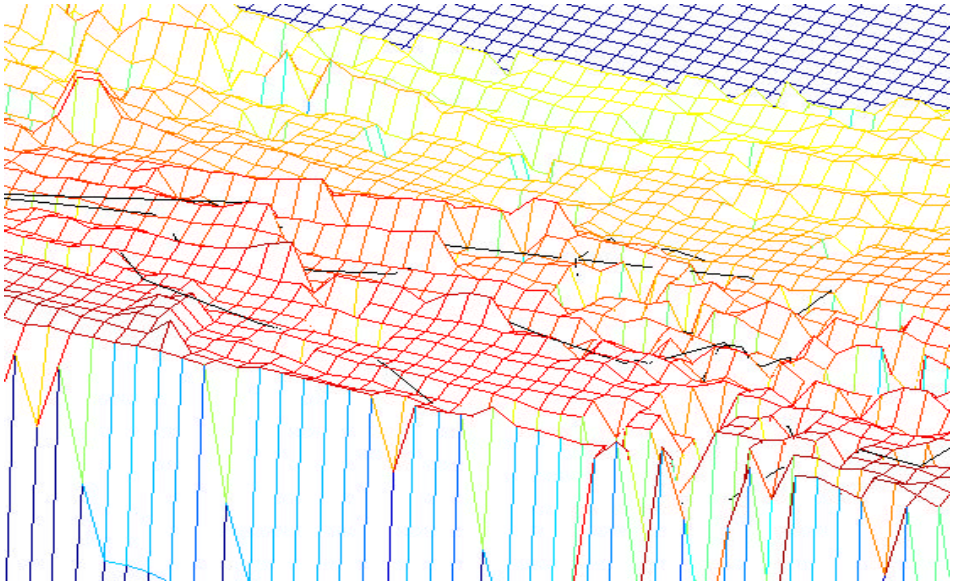

Figure 4.9 Close-up of the path on the field

Also, Figure 4.10, 4.11, and 4.12 show a complex shaped field with multiple obstacles.

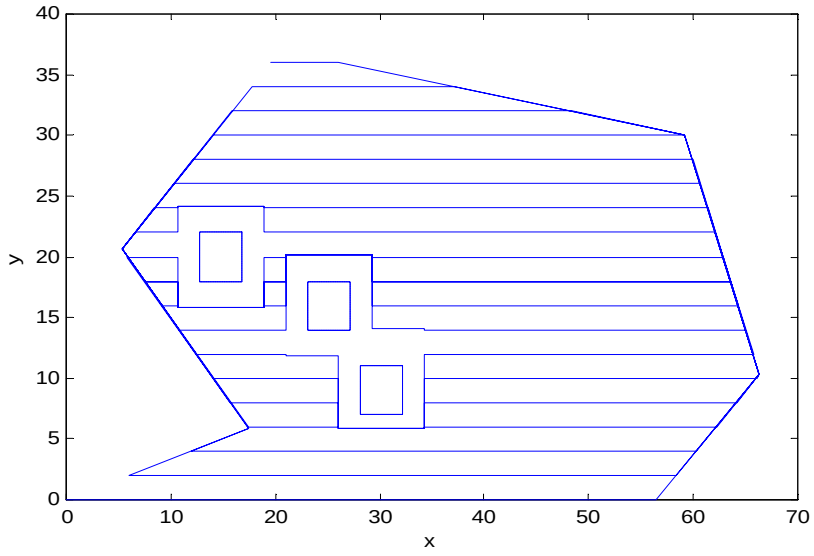

Figure 4.10. Complex field with multiple obstacles

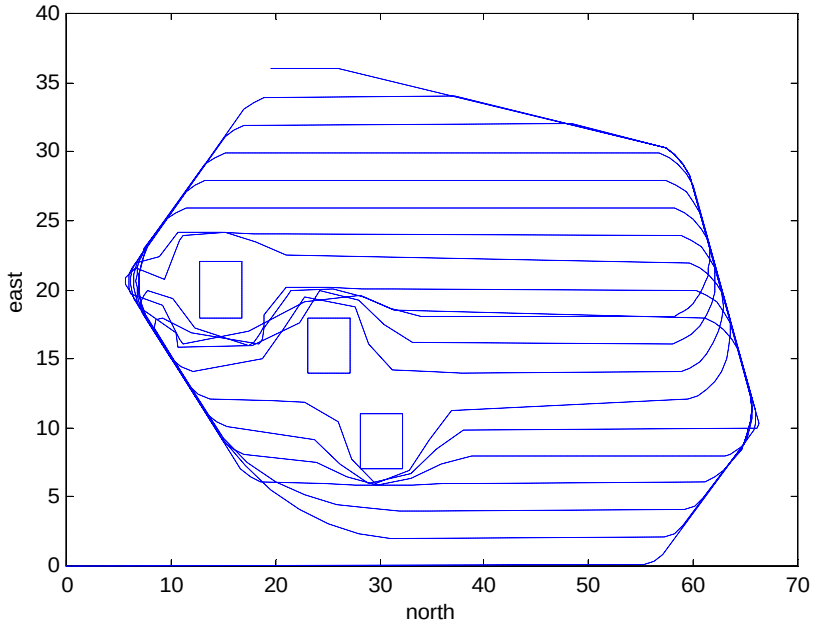

Figure 4.11 Same field after smoothing has been applied.

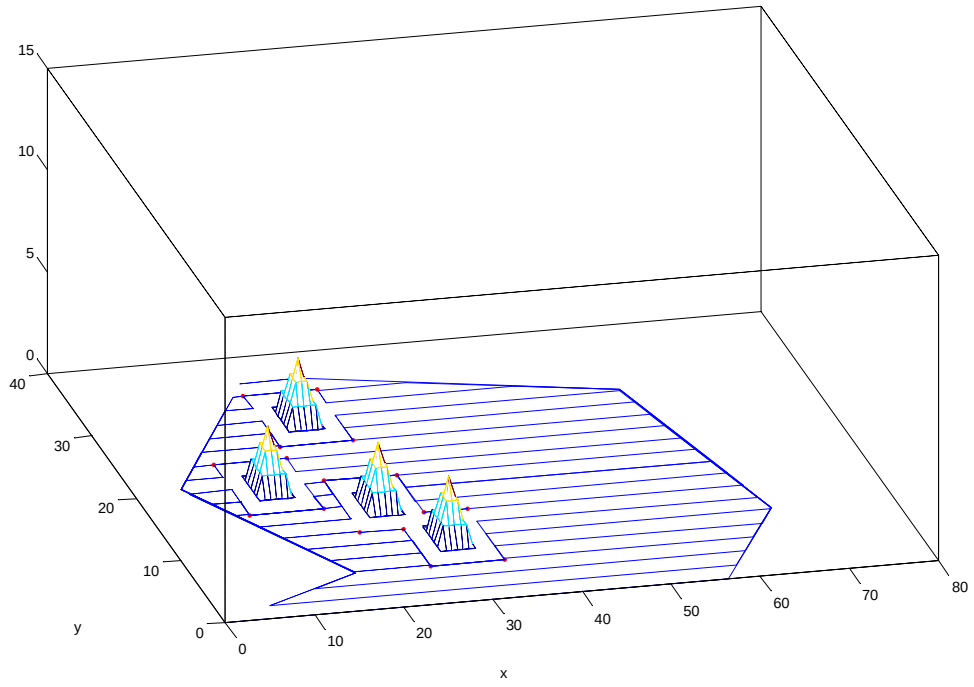

Figure 4.12 Three-dimensional view of a field.

The primary objective of this work was to develop an algorithm that calculates a survey path on a field that has any shape and that has various natural and manmade features. The algorithm presented in this paper finds the path that best suits the specified vehicle that uses it. In other words, if the terrain has features that the vehicle cannot traverse, they will be avoided. If the terrain has trees or houses they will be avoided. The implementation on the vehicle is currently in progress and it will be finalized in time. Optimization to the program constitutes work to be accomplished in the future.

CHAPTER 5
PATH PLANNER ARCHITECTURE

The Navigation Test Vehicle has one of the most efficient and modern architectures available for implementation on unmanned vehicles. The architecture has been developed at the Center for Intelligent Machines and Robotics (CIMAR) under the direction of the Air Force Research Laboratory at Tyndall Air Force Base, Panama City, FL. It consists of five self-contained modules that perform specified functions: Autonomous Control Unit (ACU), Path Planner (PLN), Position System (POS), Detection Mapping System (DMS), and Vehicle Control Unit (VCU). Figure 5.1 presents a graphical representation of the architecture. The focus of this chapter, as well as the whole paper is the PLN and its interface with the other modules, and, therefore, a detailed description will be presented. In Arm00 the authors present a detailed description of the modules that are not presented in here.

A typical interface contains messages that may be sent to the planner system and messages that will be returned. Every message is composed of a header section, a data section, and an end of message section. For each message [0x02] represents the flag for the beginning of the message and [0x03] represents the flag for the end of the message.

The header and data sections of the messages are defined here. For more details on the message formats and their components consult "2.0 Standardized Message Format" and "2.5 Path Planner (PLN)" shown in Appendix B. The possible input messages are the following: PLN Shutdown, PLN Reinitialize, PLN Set Configuration,

PLN Request Configuration, PLN Request Status, PLN Request Path DMS Report. The output messages are the following: PLN Configuration Report, PLN Status Report, PLN Path Report, DMS Start Report, DMS Stop Report. All the messages that deal with the Detection Mapping System (DMS) reports are described in the Detection Mapping System ID.

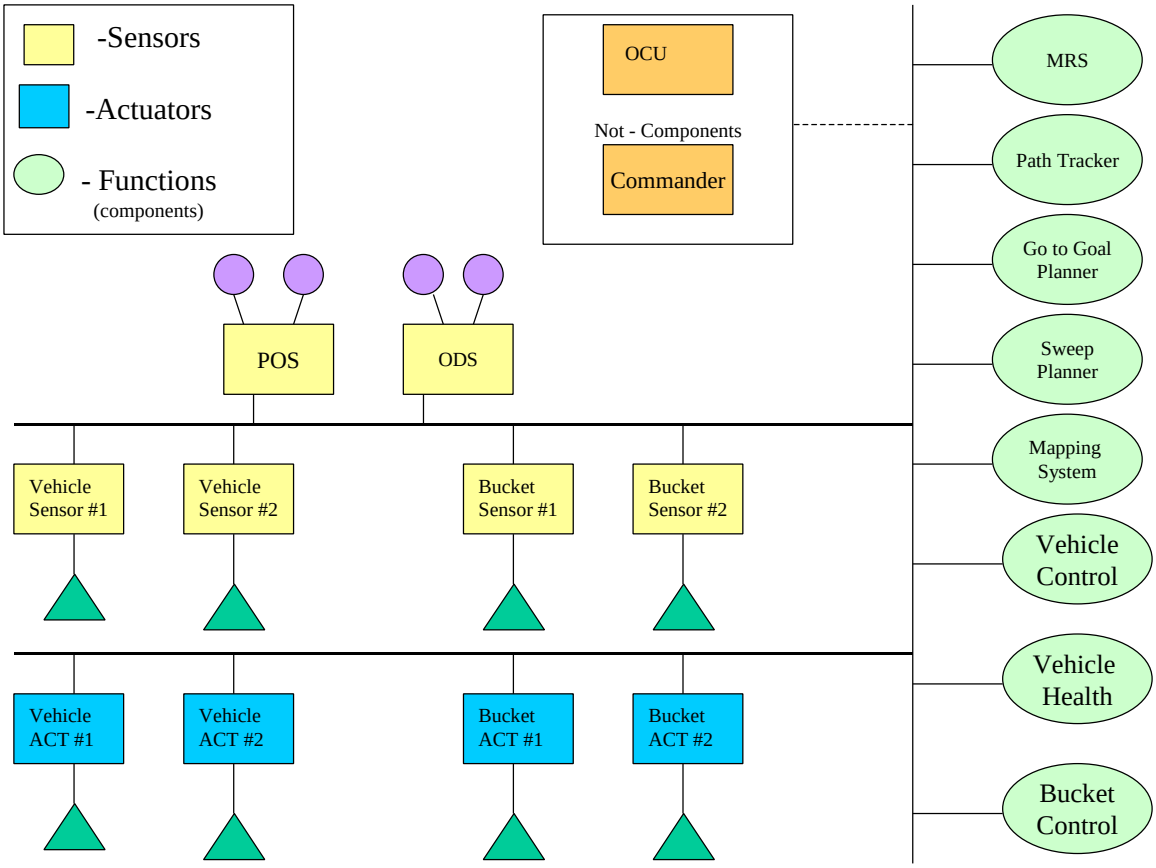

Figure 5-1. System Architecture Diagram

The PLN Shutdown message causes the planner system to shutdown all of its sub-systems in the proper fashion. At this time, the system saves any files or information that may be used on the next startup. The power to the module is then turned off. The fields

included in the header part of this message are the following: "Message ID," "Destination," "Source," "Vehicle," "Data Status," "Data Size". A sample message has the following form:

[0x02]0504,PLN,OCU,MUL,0,0,??[0x03]

The PLN Reinitialize message causes the system to restart and re-initialize all sub-systems in the proper sequence and bring the system up to a state of readiness. The fields used are the same as in PLN Shutdown. A sample message format is the following:

[0x02]0506,PLN,ACU,MUL,0,0,??[0x03]

The PLN Standby message causes the system to go into a standby mode. The fields used are the same as in the previous two messages. In standby mode the system is alive and ready to be re-initialized. A sample message looks like the following:

[0x02]0508,PLN,ACU,MUL,0,0,??[0x03]

The PLN Set Configuration message is used to set up the configuration of the system. The header section uses the same fields, while the data section includes the following fields: "Vehicle," "Length," "Width," "Height," "Turning radius," "Number of path parameters," "Parameter #1" through "Parameter #N". The parameters used are the following: distance between rows, minimum turning radius, length of the path as well as others. A sample message has the following form:

[0x02]050A,PLN,ACU,MUL,0,??,MUL,2.0,1.5,2.0,3.0,7,... ??[0x03]

The PLN Request Configuration message is used to request the current configuration of the planner system. The fields that are used in the header field are the following: "Message ID," "Destination," "Source," "Vehicle," "Data Status," and "Data

Size". There are no fields for the data section of the field. The message has the following form:

[0x02]050C,PLN,ACU,MUL,0,0,?? [0x03]

The PLN Request Status is used to request the current status of the planner system. The fields used in the header section of the message are same as in the PLN Request Configuration. Also there are no fields in the data section. The message has the following form:

[0x02]050E,PLN,ACU,MUL,0,0,??[0x03]

The PLN Request Path is used to request a path. The fields used in the header section are the same as in the PLN Request Configuration and PLN Request Status. The data section of the message ,however, contains multiple fields. They are the following: "Path Type," "# points," "Latitude #1," "Longitude #1," "Altitude #1," "theta_x (Roll)," "theta_y (Pitch)," and "theta_z (Yaw)". The last six fields can be repeated numerous times. The message has the following format:

[0x02]0520,PLN,ACU,MUL,0,??,,1,4,29.1234567,82.1234567,23.12,1.12,1.12,3.12....29.1234567...

,?[x03]

The PLN Path Report is the main message from the planner system. The header section uses the same fields as in the previous messages. Also the data section includes the same fields described in the PLN Request Path. The exception, however, is that the last spot in the data section is taken be the field "PLN Status". There are two status bytes. Status byte 1 is generic and will not change from system to system. Status byte 2 is set aside to be system specific defined by the various PLN system modules. As far as status byte 1 goes, each bit is assigned in the following manner:

0 – Startup 4 – Problem

1 – Busy	5 – Error
2 – Standby	6 – Failure
3 – Ready	7 – Shutdown

The message has the following format:

[0x02]02a0,ACU,PLN,MUL,0,??,,29.1234567,-85.1234567,23.12,
1.12,1.12,3.12,2.12,2.12,0.12,0.06,12201022,03,??[0x03]

The PLN Configuration Report is used to report the current configuration of the planner system. The header section of the message has the following fields: "Message ID," "Destination," "Source," "Vehicle," "Data Status," and "Data Size". The data section has the following fields: "Text Description" and "System Identification". A sample message looks like the following:

[0x05]05A2,ACU,PLN,MUL,0,??,Planner System: Plans go to goal and field sweep paths,11,??[0x03]

The PLN Status Report provides the host with the system status information. The header section has the same fields as the PLN Configuration Report. The data section, however, contains only one field: "Status". It has the same configuration as the status bytes described in the PLN Report. The message has the following format:

[0x02]02a4,ACU,PLN,MUL,0,8,01,??[0x03]

Since the header and data sections have been described for each message individually, the only issue left to discuss is the end of the message section. For each message, the section contains the same two fields: "Checksum" and "End of text", where the first field is represented by an eight bit unsigned sum.

There are a few comments that apply to all messages and that are essential to follow when setting up communication between various modules. All messages have to be comma delimited as it can be seen in any of the samples shown above. Data fields

that are not used can be represented just with a comma. Also, field names, such as destination and vehicle, are NOT case sensitive. The most important fact is that there can be NO white spaces in a field. Also all fields in the header must be filled.

The MAX interface documents define all of the messaging that is required of a particular module. There may be additional messaging that is available on any particular system. This additional messaging should comply with the standard message format and should be defined in the particular module's documentation. In this way you can always interchange position systems, for example, and still have all the core messaging required for communicating with it. You can also provide the user with any additional features that may be available if they so choose to use.

APPENDIX A
USEFUL WEB ADDRESSES

Formats and Documentation

Spatial Data Transfer Standard (SDTS)

http://mcmcweb.er.usgus.gov/sdts/index.html

Digital Elevation Model (DEM)

http://feature.geography.wisc.edu/sco/maps/usgsdem.html

http://feature.geography.wisc.edu/sco/maps/m_dem.html

SDTS to DEM conversions

http://geopotential.com/codes/SDTS/sdts2dem.html

http://edc.usgs.gov/nsdi/gendem.htm

Applications for DEM's

3DEM

http://www.mnsinc.com/rshorne/3dem.html

DEM Tools

www.arq.net/~karsten/demtools

dlgv32

http://mcmcweb.er.usgs.gov/viewers/dlg_view.html

Leveller

www.daylongraphics.com/products/leveller

Microdem

www.usna.edu/Users/oceano/pguth/website/microdem.html

Terragen

www.planetside.co.uk/terragen

Wilbur

www.ridgenet.net/~jslayton/software.html

APPENDIX B
DEM FORMAT

Data Records

Record type A, element 1 has been changed to require certain information in specified byte locations. A new element, element 2, record type A, has been defined to record the NMD organization from which the DEM was authorized. The element counts of old record type A, elements 2-15, have been incremented by one to elements 3-16 An attempt has been made to keep the old and new DEM formats compatible; therefore, although the element counts have changed, the byte positions and information content of these fields (old elements 2-15, new elements 3-16) remain the same. This change should be transparent to old DEM applications programs. The new element 2, the mapping center of DEM origin is named in record A, bytes 141-144. Valid codes are MAC (Mapping Application Center, GPM2 (specific to MAC Gestalt Photo Mapper II auto correlator), MCMC (Mid Continent Mapping Center), RMMC (Rocky Mountain Mapping Center), WMC (Western Mapping Center) and FS (U.S. Forest Service). Codes indicating other sources of DEM's are defined when required. Also, new data elements 17-29 have been appended to the end of the type A record. These elements are contained in the end of the previously blank filled portion of the 1,024 byte record.

The type A record contains information defining the general characteristics of the DEM, including descriptive header information relating to the DEM's name, boundaries, units of measurement, minimum and maximum data values, number of type B records, and projection parameters. There is only one type A record for each DEM file, and it appears as the first record in the data file. The type B record contains elevation data and associated header information. All type B records of the DEM files are made up of data from one-dimensional bands called profiles. Therefore, the number of complete profiles covering the DEM area is the same as the number of type B records in the DEM. In a UTM structured DEM, an occasional profile exists within the bounds of the DEM quadrilateral but is void of elevation grid points and is not represented in the DEM. (This is called the "missing profile condition" and occurs occasionally as the first or last hypothetical profile of the DEM at the respective DEM corner.) The type C record contains statistics on the accuracy of the data in the file.

The following special conventions shall be observed for the population of data fields in the A, B, and C record elements:

- All character fields must be in upper case. Character fields of no data value must be blank, ASCII space (binary 0010 0000).
- All integer or character flagged fields of no data value but which default to zero must be ASCII zero (binary 0011 0000).

- All real (non-integer) numeric fields shall be populated. Default zero fill shall follow the following convention:

```
 12345678901234567890123456 (byte position, left justified)
"   .000000000000000D+00"| Standard format specified is
"  0.0"                  | D24.15. Zero values listed are
"  0.000000000000000D+00"| common machine dependent numeric
"   .000000000000000"    | defaults for zero values.
```

Type A Record

Sample DEM Type A Logical Records-Reno, Nevada-California, Quadrangle (West Half) (1 degree)

```
NJ11-01W
            3    1   0    0   0.0           0.0       0.0      0.0
0.0
              0.0                    0.0          0.0      0.0       0.0
      0.0                 0.0                  0.0               0.0    3
2    4  -0 42840000000000D+06   0.140400000000000D+06  0.428400000000000D+.06
     0.144000000000000D+06   -0.424800000000000D+06   0.144000000000000D+06
     -042480000000000D+06  0.140400000000000D+06   0.999000000000D+03
     0.264100000000000D+04  0.0   00.300000E+010.300000E+010.100000E+01  1
     1201
```

Data Element	Content	Explanation
1	NJ11-01W	Quadrangle name field (144 characters); NJ11-01W is the designation for the DEM covering the west half of the Reno, Nevada-California, sheet.
2	3	DEM level code; 3 reflects processing by DMA which includes registration to planimetric features appearing on the source 1-degree.
3	1	Pattern code; 1 indicates a regular elevation pattern.
4	0	Planimetric reference system code; 0 indicates geographic coordinate system.
5	0	Zone code; there are no zones in the geographic system. Therefore, the zone code is set to zero.
6	0.0 (15 sets of 0.0)	Map projection parameters; all 15 fields are set to zero for the geographic system and should be ignored. Presence of non-zero parameters are not related to the geographic coordinate system and should also be ignored.
7	3	Units code; 3 represents arc-seconds as the unit of

		measure for ground planimetric coordinates throughout the file.
8	2	Units code; 2 represents meters as the unit of measure for elevation coordinates throughout the file.
9	4	Number (n) of sides in the polygon which defines the coverage of the DEM file (usually equal to 4).
10	-0.4284...D+06, 0.1404...D+06 .... -0.4248...D+06 0.1404...D+06	A 4,2 array containing the ground coordinates of the four corners of the DEM. In this case translation from arc-seconds to degrees, minutes and seconds yields: -119 00 0, 39 00 00; -119 00 00, 40 00 00; -118 00 00, 40 00 00; -118 00 00. 39 00 00.
11	0.9990...D+06 0.2641...D+04	Minimum and maximum elevations for the DEM.
12	0.0	Counterclockwise angle from the primary axis of ground planimetric referenced to the primary axis of the DEM local reference system.
13	0	Accuracy code; 0 indicates that a record of accuracy does not exist and that no record type C will follow.
14	0.300000E+01 0.300000E+01 0.100000E+01	A three-element containing DEM spatial resolution (x,y, z); set to 3, 3, 1; or 3, 6, 1; or 3, 9, 1 depending on the latitude of the product.
15	1 1201	A two-element array containing the number of rows and columns of profiles in the DEM. The row value m is usually set to 1 as an indication that the arrays are actually one-dimensional profiles. The column value n set to 1,201 indicates that the DEM file consists of a total of 1,201 profiles.

Type B Record

DEM Type B Logical Record-Reno, Nevada California, Quadrangle (West Half) (1 degree)

```
     1      1   1201    1  -0.42480000000000D+06   0.1404000000000D+06  0.0
0.121000000000D+04  0.177200000000D+04  1538  1539  1539  1534  1529  1520
1513   1508  1501  1493  1488  1479  1475  1474  1473  1472  1469  1467
1466
1464
```

Data Element	Content	Explanation
1	1 1	Row and column identification number of the profile contained in this record; 1, 1 represents row 1, column 1, in the DEM data set.
2	1201 1	Number of rows (elevations) and columns in this profile; 1201, 1 indicates there are 1,201 elevations and 1 column in this profile.
3	-0.4284...D+06 0.1404...D+06	Translated to the decimal, -428400.0 and 140400.0 are the ground planimetric coordinates (arc-seconds) of the first elevation in the profile, thus computed equal to -119 and 39 degrees.
4	0.0	Elevation of local datum for the profile. Always zero for 1-degree DEM, the reference is mean sea level.
5	0.1212...D+04 0.1772...D+04	Minimum and maximum elevations for the profile.
6	1538 1539...	An array of m x n elevations (m=1201, n=1) for the profile expressed in units of measure of meters (record A, element 8, indicates meters as units of measure) above or below the local datum (record a, element 4).

Type C Record

Elevation Model Data Elements Logical Record Type C

Data Element	Type (FORTRAN Notation)	Physical Record Format: ASCII Format	Starting byte	Ending byte	Comment
1 Code indicating availability of statistics in data element 2	INTEGER*2	16	1	6	Code 1 = available 0 = unavailable
2 RMSE of file's datum relative elements to absolute datum (x, y, z)	INTEGER*2	3I6	7	24	In some units as indicated by 8 and 9 of logical record type A.
3 Sample size on which statistics be	INTEGER*3	I6	25	30	If 0, then accuracy will be assumed to

	in data element than 2 are based					estimated rather than computed.
4	Code indicating availability of statistics in data element 5	INTEGER*2	I6	31	36	Code 1 = available 0 = unavailable
5	RMSE of DEM data relative elements to file's datum (x, y, z)	INTEGER*2	3I6	37	54	In same units as indicated by 8 and 9 of logical record type A.
6	Sample size on which statistics in element 5 are based	INTEGER*2	I6	55	60	If 0, then accuracy will be assumed to be estimated rather than computed.

APPENDIX C
FILE FORMATS

VEHICLE_DATA FORMAT

```
# vehicle_struct data file
l 1.5
w 1
h 4.1
t 7.5
*
```

MAP_DATA FORMAT

```
# this file contains virtual obstacles based on bad slopes

m 33


b 4


n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
n 0 0 1 2 3 0.200000 4 5 6 4 0.000000
```

```
N

#vertex data

s 0 0 0.0 -0.707

S

g 999 999

G

p
16.500000 0.500000
16.500000 -0.500000
15.500000 -0.500000
15.500000 0.500000

p
20.500000 0.500000
20.500000 -0.500000
19.500000 -0.500000
19.500000 0.500000

p
24.500000 0.500000
24.500000 -0.500000
23.500000 -0.500000
23.500000 0.500000

p
28.500000 0.500000
28.500000 -0.500000
27.500000 -0.500000
27.500000 0.500000

p
32.500000 0.500000
32.500000 -0.500000
31.500000 -0.500000
31.500000 0.500000

p
36.500000 0.500000
36.500000 -0.500000
35.500000 -0.500000
```

35.500000 0.500000

p
40.500000 0.500000
40.500000 -0.500000
39.500000 -0.500000
39.500000 0.500000

p
44.500000 0.500000
44.500000 -0.500000
43.500000 -0.500000
43.500000 0.500000

p
48.500000 0.500000
48.500000 -0.500000
47.500000 -0.500000
47.500000 0.500000

p
52.500000 0.500000
52.500000 -0.500000
51.500000 -0.500000
51.500000 0.500000

p
56.500000 0.500000
56.500000 -0.500000
55.500000 -0.500000
55.500000 0.500000

p
57.536787 0.500000
57.536787 -0.500000
56.536787 -0.500000
56.536787 0.500000

p
56.459249 4.352131
56.459249 3.352131
55.459249 3.352131
55.459249 4.352131

p
6.520248 9.209562

6.520248 8.209562
5.520248 8.209562
5.520248 9.209562

p
4.245822 5.919123
4.245822 4.919123
3.245822 4.919123
3.245822 5.919123

p
15.264891 4.500000
15.264891 3.500000
14.264891 3.500000
14.264891 4.500000

p
19.264891 4.500000
19.264891 3.500000
18.264891 3.500000
18.264891 4.500000

p
23.264891 4.500000
23.264891 3.500000
22.264891 3.500000
22.264891 4.500000

p
27.264891 4.500000
27.264891 3.500000
26.264891 3.500000
26.264891 4.500000

p
31.264891 4.500000
31.264891 3.500000
30.264891 3.500000
30.264891 4.500000

p
35.264891 4.500000
35.264891 3.500000
34.264891 3.500000
34.264891 4.500000

p
39.264891 4.500000
39.264891 3.500000
38.264891 3.500000
38.264891 4.500000

p
43.264891 4.500000
43.264891 3.500000
42.264891 3.500000
42.264891 4.500000

p
47.264891 4.500000
47.264891 3.500000
46.264891 3.500000
46.264891 4.500000

p
51.264891 4.500000
51.264891 3.500000
50.264891 3.500000
50.264891 4.500000

p
55.264891 4.500000
55.264891 3.500000
54.264891 3.500000
54.264891 4.500000

p
56.417887 4.500000
56.417887 3.500000
55.417887 3.500000
55.417887 4.500000

p
7.010713 9.919123
7.010713 8.919123
6.010713 8.919123
6.010713 9.919123

p
6.029783 8.500000
6.029783 7.500000
5.029783 7.500000

5.029783 8.500000

p
10.029783 8.500000
10.029783 7.500000
9.029783 7.500000
9.029783 8.500000

p
14.029783 8.500000
14.029783 7.500000
13.029783 7.500000
13.029783 8.500000

p
30.029783 8.500000
30.029783 7.500000
29.029783 7.500000
29.029783 8.500000

p
50.029783 8.500000
50.029783 7.500000
49.029783 7.500000
49.029783 8.500000

P

q
-1000 -1000
-1000 1000
1000 1000
1000 -1000

*
*

APPENDIX D
MESSAGE FORMATS

2.5 Software Interface Document - Path Planner (ASCII)

Version 2.0

This section presents the messages that may be sent to the planner system and the messages that will be returned. Every message will be composed of a header section, a data section, and an end of message section. The data section of the messages are defined here. See the standardized message format documentation (section 2.0) for the header and end of message formats.

Note: See specific module documentation for additional (system specific, non required) messages.

I. Input Messages:

- PLN Shutdown - 0x0504
- PLN Reinitialize - 0x0506
- PLN Set Config. - 0x050A
- PLN Request Config. - 0x050C
- PLN Request Status - 0x050E

- PLN Request Path - 0x0520

- DMS Report - See Detection Mapping System ID

II. Output Messages:

- PLN Config. Report - 0x05A2
- PLN Status Report - 0x05A4

- PLN Path Report - 0x05C0

- DMS Start Report - See Detection Mapping System ID
- DMS Stop Report

PLN Shutdown

Input Message

The PLN Shutdown message causes the planner system to shutdown all of its sub-systems in the proper fashion. At this time, the system may save any files or information that may be used on the next startup. The power to the module may then be turned off.

Header section:

Field#	Name	Description	Example
1	Message ID		0504
2	Destination		PLN
3	Source	Set by the host system (ACU, OCU)	ACU
4	Vehicle	Set using the set Config msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Number of bytes in data (decimal) Exclude the leading & trailing ›,:	0

Data Section

Field#	Name	Description	Example

NO DATA

Sample Message:

[0x05]0504,PLN,OCU,MUL,0,0,??[0x03]

PLN Reinitialize

Input Message

The PLN Reinitialize message causes the system to restart and re-initialize all sub-systems in the proper sequence and bring the system up to a state of readiness.

Header section:

Field#	Name	Description	Example
1	Message ID		0506
2	Destination		PLN
3	Source	Set by the host system (ACU, OCU)	ACU
4	Vehicle	Set using the set Config msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Number of bytes in data (decimal) Exclude the leading & trailing ›;:	0

Data Section

Field#	Name	Description	Example

Sample Message:

[0x05]0506,PLN,ACU,MUL,0,0,??[0x03]

PLN Standby

Input Message

The PLN Standby message causes the system to go into a standby mode. In standby mode the system is alive and ready to be re-initialized.

Note: A PLN Shutdown should be given prior to turning system power off.

Header section:

Field#	Name	Description	Example
1	Message ID		0508
2	Destination		PLN
3	Source	Set by the host system (ACU, OCU)	ACU
4	Vehicle	Set using the set Config msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Number of bytes in data (decimal) Exclude the leading & trailing ›,:	10

Data Section

Field#	Name	Description	Example

NO DATA

Sample Message:
[0x02]0508,PLN,ACU,MUL,0,0,??[0x03]

PLN Set Configuration

Input Message

The PLN Set Configuration message is used to set up the configuration of the system. The following gives an explanation of the parameters:

Header section:

Field#	Name	Description	Example
1	Message ID		050A
2	Destination		PLN
3	Source	Set by the host system (ACU, OCU)	ACU
4	Vehicle	Set using the set Config msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Variable, set prior to shipping	??

Data Section

Field#	Name	Description	Example
7	Vehicle	The system that the module is physically contained in.	MUL
8	Length	Lenght of the vehicle (Meters)	2.0
9	Width	Width of the vehicle (Meters)	1.5
10	Height	Height of the vehicle (Meters)	2.0
11	Turning radius	Vehicles minimum turning radius (Meters) Note: If the vehicle is omnidirectional then this variable should be set to zero.	3.0
12	Number of path parameters	The number of path planning parameters, that are specific to a given path planner, in the current message.	7
13	Parameter #1	N parameters that are specific to a given path planner, (e.g., use a boundary, restrict start, row distance, etc...).	
14	Parameter N		

Sample Message:

```
[0x05]050A,PLN,ACU,MUL,0,??,MUL,2.0,1.5,2.0,3.0,7,...  ??[0x03]
```

PLN Request Configuration

Input Message

This message is used to request the current configuration of the planner system. See PLN Configuration Report for the response definition.

Header section:

Field#	Name	Description	Example
1	Message ID		050C
2	Destination		PLN
3	Source	Set by the host system (ACU, OCU)	ACU
4	Vehicle	Set using the setConfig msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Number of bytes in data (decimal) Exclude the leading & trailing ›;:	0

Data Section

Field#	Name	Description	Example

NO DATA

Sample Message:

[0x05]050C,PLN,ACU,MUL,0,0,?? [0x03]

PLN Request Status

Input Message

This message is used to request the current status of the planner system. See PLN Status Report for the response definition.

Header section:

Field#	Name	Description	Example
1	Message ID		050E
2	Destination		PLN
3	Source	Set by the host system (ACU, OCU)	ACU
4	Vehicle	Set using the set Config msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Number of bytes in data (decimal) Exclude the leading & trailing ›,:	0

Data Section

Field#	Name	Description	Example

NO DATA

Sample Message:

[0x05]050E,PLN,ACU,MUL,0,0,??[0x03]

PLN Request Path

Input Message

This message is used to request a plan. The following is a definition of the message parameters. See PLN Path Report for the response report definition.

Header section:

Field#	Name	Description	Example
1	Message ID		0520
2	Destination		PLN
3	Source	Set by the host system (ACU, OCU)	ACU
4	Vehicle	Set using the set Config msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Number of bytes in data (decimal) Exclude the leading & trailing ›,:	??

Data Section

Field#	Name	Description	Example
7	Path Type	The path type describes what kind of plan is being requested. For example you may request a go to goal path and give the start and goal poses. Alternatively, you may request a sweep path and specify the corner points of the field. See the Path Type table for type identifiers.	1
8	# points	The number of points specified These would be the start, goal, corner points etc.	4
9	Latitude #1	Latitude of the first point in WGS-84 geodetic coordinates (Deg.dec_deg)	29.1234567
10	Longitude #1	Longitudeof the first point in WGS-84 geodetic coordinates. (Deg.dec_deg)	-85.1234567
11	Altitude #1	Altitude of the first point meters	23.12
12	theta_x (Roll)	Orientation about the x axis at the first point radians -PI/2.0 to PI/2.0 Use the right hand rule, x is forward, z is down (axis attached to vehicle)	1.12
13	theta_y (pitch)	Orientation about the y axis at the first point radians -PI/2.0 to PI/2.0	1.12

14	theta_z (yaw)	Orientation about the z axis at the first point radians 0 to 2.0 x PI 0 = Geodetic North	3.12
?	Latitude #n		29.1234567
?	Longitude #n		-85.1234567
?	Altitude #n		23.12
?	theta_x (Roll)		1.12
?	theta_y (pitch)		1.12
?	theta_z (yaw)		3.12

Sample Message:
[0x05]0520,PLN,ACU,MUL,0,??,,1,4,29.1234567,82.1234567,23.12,1.12,1.12,3.12....29
.1234567... ,?[x03]

Path Type Table

Path Type #	**Description**	**Required Information**
0		
1	Go to Goal	Start Position Goal Position
2		
3		
4		
5	Field Sweep	Corner points of field

PLN Path Report

Output Message

This is the main message from the planner system. The following is a description of the parameters:

Header section:

Field#	Name	Description	Example
1	Message ID		05C0
2	Destination	Destination will be set equal to the source of the startReport message	ACU
3	Source		PLN
4	Vehicle	Set using the set Config msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Variable, set prior to shipping (use sizeof)	??

Data Section

Field#	Name	Description	Example
7	Path Type	See table of path types under PLN Request Plan	1
8	Path Status	0=Path OK 1=Path is not valid	0
9	Path Length	Meters	2500
10	# subgoals	the number of subgoals	950
11	Latitude #1	Latitude of the first point in WGS-84 geodetic coordinates (Deg.dec_deg)	29.1234567
12	Longitude #1	Longitude of the first point in WGS-84 geodetic coordinates. (Deg.dec_deg)	-85.1234567
13	Altitude #1	Altitude of the first point meters	23.12
14	theta_x (Roll)	Orientation about the x axis at the first point radians -PI/2.0 to PI/2.0 Use the right hand rule, x is forward, z is down (axis attached to vehicle)	1.12
15	theta_y (pitch)	Orientation about the y axis radians -PI/2.0 to PI/2.0	1.12
16	theta_z (yaw)	Orientation about the z axis radians 0 to 2.0 x PI 0 = Geodetic North	3.12
??	Latitude #n		29.1234567

??	Longitude #n		-85.1234567
??	Altitude #n		23.12
??	theta_x (Roll)		1.12
??	theta_y (pitch)		1.12
??	theta_z (yaw)		3.12
??	PLN Status	See Below	03

Example Message:
[0x05]02a0,ACU,PLN,MUL,0,??,,29.1234567,-85.1234567,23.12,
1.12,1.12,3.12,2.12,2.12,0.12,0.06,12201022,03,??[0x03]

Status Byte Description:
Status byte 1 is generic and will not change from system to system.
Status byte 2 is set aside to be system specific defined by the various PLN system modules.

Status Byte 1		Status Byte 2	
Bit VCU	Condition when set 1 = set	Bit VCU	Condition when se 1=set
0	Startup	0	Contractor Reserved
1	Busy	1	Contractor Reserved
2	Standby	2	Contractor Reserved
3	Ready	3	Contractor Reserved
4	Problem	4	Contractor Reserved
5	Error	5	Contractor Reserved
6	Failure	6	Contractor Reserved
7	Shutdown	7	Contractor Reserved

Description:

Startup: Indicates the system has just been powered up

Busy: Indicates the system is currently processing the last command

Standby: Indicates the system is ready to be reinitialize

Ready: Indicates that the system is initialized and is operational

Problem: Indicates that a self-correcting problem has occurred and the problem is being corrected internally. This problem requires no input from the host

Error: Indicates that a problem has occurred that the system could not resolve. An error requires the intervention of the host to be resolved.

Failure: Indicates that the system has failed and will not recover.

PLN Configuration Report

Output Message

This message is used to report the current configuration of the planner system. The following is a description of the parameters:

Header section:

Field#	Name	Description	Example
1	Message ID		05A2
2	Destination	Destination will be set equal to the source of the startReport message	ACU
3	Source		PLN
4	Vehicle	Set using the set Config msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Variable, set prior to shipping (use sizeof)	??

Data Section

Field#	Name	Description	Example
7	Text Description	Free form text description. May list the main components used by the system and or other pertinent information.	Planner System: bla bla bla
8	System Identification	Gives a hex number assigned to the particular planner system so that it may be more uniquely described.	11

Sample Message:

```
[0x05]05A2,ACU,PLN,MUL,0,??,Planner System: bla bla bla,11,??[0x03]
```

PLN Status Report

Output Message

Provides the host with the system status information

Header section:

Field#	Name	Description	Example
1	Message ID		05A4
2	Destination	Destination will be set equal to the source of the start Report message	ACU
3	Source		PLN
4	Vehicle	Set using the set Config msg.	MUL
5	Data Status	See section 2.0, Standardized message format	0
6	Data Size	Number of bytes in data (decimal) Exclude the leading & trailing ›,:	8

Data Section

Field#	Name	Description	Example
7	Status	4 bytes: See Below	01

Example Message:
[0x05]02a4,ACU,PLN,MUL,0,8,01,??[0x03]

7. See PLN Report for Status Byte Description

APPENDIX E
COMPUTER CODE

```
#include<stdlib.h>
#include<stdio.h>
#include<math.h>
#include<time.h>
#include <stdio.h>
#include <math.h>
#include "fieldSweepLib.h"
#include "ufMathLib.h"
#include "includes.h"

#define SIZE                  70          /* size of grid SIZE X SIZE
*/
#define OK                    1           /* true value */
#define TOL                   0.001       /* tolerance */
#define DIM                   5           /* distance between any two
                                          points in the grid */
#define CRITICAL_SLOPE        5
#define M_PI                  3.14159265359
#define D2R                   M_PI/180
#define R2D                   180/M_PI
#define EARTH_RADIUS          6.378137e6
#define deltaX                0.5
#define deltaY                0.5
#define REGULAR               1
#define MIN                   0
#define GOOD                  1
#define BAD                   0
#define MINX                  0.00                    /* 29.646274 */
#define MAXX                  69.897508               /* 29.646901 */
#define MINY                  -32.110427              /* -82.353936 */
#define MAXY                  11.967604               /* -82.353480 */
#define noColumns             44
#define noRows                69

double heightMesh(double *field, int i, int j, double s, double t);
void corners(double x, double y, int *iWanted, int *jWanted);
void pAndR(double *field,double x, double y, double *pitch, double
*roll, double *length);
void checkHeights(double, double, double, double, double, double,
double, double, double *field);
void extractRealData(int);

void main(void)
{
    /* create the field */

      double *grid;
```

```
	double init[4][2], Pts[4][2];
	double minCost, north, east, alt;
	char ch, dummy;
	int i,j ;
	double d = DIM+1, ang, costSum, deltaTheta, minAngle;
	int e,s;double timeSum;
	int points;
	float dist_rows,minTurnRad;
	struct _geoPoint *plan=NULL, *tmp;

	FILE *fpHeight, *fpPoints, *fpTerrain, *fpRealHeight, *fpHeight2;

	if ((grid = (double *)(malloc(sizeof(double) * ((noColumns+1) *
	       (noRows+1)))))
	      == NULL){
	      free(grid);
	      printf("error allocating memory!!!\n");
	      return(-1);
	}

	/* initialize the height grid with values from the converted DEM
	 "zcut.txt" contains 1024 values for height*/
	if ((fpHeight = fopen("zcut.txt","w")) == NULL) {
	      printf("error opening 'zcut.txt' file\n") ;
	      exit(1) ;
	}

	for(i=0 ; i<noRows ; i++)
	      for(j=0 ; j<noColumns ; j++)
	            /* field[i][j] = 0.1*(-1.0 + 2.0*rand()/RAND_MAX) ;
*/
	            /* fscanf(fpHeight,"%lf", &field[i][j]); */
	            //grid[i + j*noColumns] = -99.9;
	            grid[i + j*noColumns] = 10.0;

	/* initialize the 2D path planning algorithm */
	printf("What is the distance between rows?\n");
	scanf("%f",&dist_rows);

	printf("What is the minimum turning radius \n");
	scanf("%f",&minTurnRad);

	printf("What is the angle for the sweep\n");
	scanf("%lf", &deltaTheta);

	points= 4;        /* number of corner points for the field-grid
*/

	/* define grid for a fixed number of data points */
	i=0 ; j=0;
	init[0][0] =  i ;
	init[0][1] =  j ;
	init[1][0] =  SIZE*DIM;
	init[1][1] =  j ;
	init[2][0] =  SIZE*DIM ;
	init[2][1] =  SIZE*DIM ;
	init[3][0] =  i ;
```

```
      init[3][1] =  SIZE*DIM ;

      printf("Please introduce the number of points defining the
field:")
      scanf("%d", &points);

      /* introduce the corners for the needed field */
      printf("Please introduce the coordinates for the field corners(x,
y):\n");
      printf("Note: points are in local coordinates\n");
      scanf("%d %d\n", &init[0][0],&init[0][1]);
      scanf("%d %d\n", &init[1][0],&init[1][1]);
      scanf("%d %d\n", &init[2][0],&init[2][1]);
      scanf("%d %d\n", &init[3][0],&init[3][1]);

      /* convert to lat/lon coordinates for link to the 2D algorithm */
      for(i=0; i<4 ; i++){
            Pts[i][0] = init[0][0] + (init[i][0] / EARTH_RADIUS) * R2D
;
            Pts[i][1] = init[0][1] +(init[i][1] / (EARTH_RADIUS *
cos(init[0][0]*D2R))) * R2D ;
      }

      /* coordinates for test field */
      /* should be used if the file with the heights is used as well */
      Pts[0][0] = 29.6463086;
      Pts[0][1] = -82.3538956;
      Pts[1][0] = 29.6468305;
      Pts[1][1] = -82.3539348;
      Pts[2][0] = 29.6467917;
      Pts[2][1] = -82.3537752;
      Pts[3][0] = 29.646455;
      Pts[3][1] = -82.3536519;

      /* use if given a file with test data recorded by the mule */
      /* extractRealData(5); */

      if ((fpTerrain = fopen("terrain.txt","r")) == NULL){
            printf("Can't open 'terrain.txt' file \n");
            /* in case that there is no file exit */
            exit(1);
      }

      /* read north, east, and altitude data from file */
      while(fscanf(fpTerrain,"%lf %lf %lf\n", &north, &east, &alt) ==
3) {
            grid[(int)(north-MINX) + (int)(east-MINY)*noColumns] = alt;
      }

      fclose(fpTerrain);

      for(i=0 ; i < noRows ; i++){
            for(j=0 ; j < noColumns ; j++)
                  fprintf(fpHeight,"%6.2f  ", grid[i + j*noColumns]);
            fprintf(fpHeight,"\n");
      }
      fclose(fpHeight);
```

```
      for(i=1; i<noRows-1; i++)
            for(j=1; j<noColumns-1; j++)
                  if((grid[i+j*noColumns] > (10.0-TOL)) &&
(grid[i+j*noColumns]                        < (10.0+TOL)))
                        grid[i+j*noColumns] = (grid[(i-1)+j*noColumns]
+                                 grid[i+(j-1)*noColumns] +
                              grid[(i+1)+j*noColumns] +
                                    grid[i+(j+1)*noColumns])/4;

      if ((fpHeight2 = fopen("zcutModified.txt","w")) == NULL) {
            printf("error opening 'zcutModified.txt' file\n") ;
            exit(1) ;
      }

      for(i=0 ; i < noRows ; i++){
            for(j=0 ; j < noColumns ; j++)
                  fprintf(fpHeight2,"%6.2f  ", grid[i + j*noColumns]);
            fprintf(fpHeight2,"\n");
      }
      fclose(fpHeight2);

/*
      if ((fpRealHeight = fopen("realHeightCheck.txt","w")) == NULL){
            printf("Can't open 'realHeightCheck.txt' file \n");
            /* in case that there is no file exit */
/*          exit(1);
      }

      for(i=0 ; i<SIZE ; i++){
            for(j=0 ; j<SIZE ; j++)
                  fprintf(fpRealHeight,"%6.2f  ", field[i][j]);
                  fprintf(fpRealHeight,"\n");
      }
      fclose(fpRealHeight);
*/
      while(d > DIM){
            printf("Distance between points for cost
calculations(smaller than %d):",DIM);
            scanf("%lf", &d);

            if(d > DIM)
                  printf("\ndistance has to be smaller than %d for
                  reasonable results\n\n",DIM);
      }

      printf("\nCalculations in progress.  Please Wait...\n");

      minCost = 99999999;
      minAngle = 0.0;
      s=clock();
      for(ang=0; ang<180.0*D2R; ang+=deltaTheta*D2R)
      {

      fieldSweep(ang,grid,d,points,Pts,dist_rows,minTurnRad,&plan,
&costSum, REGULAR);
            if(costSum<minCost){
```

```
                minCost=costSum;
                minAngle=ang;
            }
        }
        e=clock();
        timeSum=((double)(e-s))/1000.;
        printf("\n\ntime = %.2lf sec at %.2lf deg at a cost = %.2lf for
minimum path\n",
                timeSum, minAngle*R2D, minCost);

        /* printf("the minimum cost for a path is %lf at %lf degrees\n",
minCost, minAngle*R2D); */

        /* create the minimum path based on the above values */
        printf("\nsearching for non-traversable regions...\n");
        fieldSweep(minAngle,grid,d,points,Pts,dist_rows,minTurnRad,&plan,
&costSum, MIN);

        /* path pts in lat/lon coordinates are kept in "outPoints.txt" */
        /* path pts in local coordinates are kept in "world_order.txt" */
        fpPoints = fopen("outPoints.txt","w");

        tmp = plan ;
        while (tmp != NULL) {
            fprintf(fpPoints,"o %lf %lf %lf\n",tmp->lat , tmp->lon) ;
            tmp = tmp->next ;
        }
        fprintf(fpPoints,"*\n") ;
        fclose(fpPoints) ;

        /* free memory from the field */
        free(grid);

        /* free memory from plan */
        while (plan != NULL) {
            tmp = plan ;
            plan = plan->next ;
            free(tmp) ;
        }
}

/* calculates the cost for a specified path based on pitch , roll, and
length of the path */
double cost(double *field, double x, double y, double d, int value, int
*unknown)
{
        double pitch, roll, length;
        double costPitch, costRoll, costLength, totalCost;

        pAndR(field, x, y, &pitch, &roll, &length);

        /* calculate pitch cost */
        if(fabs(pitch) <= CRITICAL_SLOPE)
            costPitch = fabs(pitch) / CRITICAL_SLOPE;
        else if(fabs(pitch) > CRITICAL_SLOPE){
            costPitch = 9999;
```

```
            if(value == 0){
                  *unknown = BAD;
            }
      }

      /* calculate roll cost */
      if(fabs(roll) <= CRITICAL_SLOPE)
            costRoll = fabs(roll) / CRITICAL_SLOPE;
      else if(fabs(roll) > CRITICAL_SLOPE){
            costRoll = 9999;

            if(value == 0){
                  *unknown = BAD;
            }
      }

      /* calculate length cost */
      costLength=d/(cos(pitch))/DIM;

      totalCost=costPitch+costRoll+costLength;

      return(totalCost);
}

/* calculate the height at a specified point in a mesh generated
between four points */
/* s and t are the LOCAL coordinate system */
double heightMesh(double *field, int i, int j, double x, double y)
{
      double C1, C2, C3, C4, specifiedHeight;
      double s, t ;

      s = x - (double)i*DIM ;
      t = y - (double)j*DIM ;

/*            t
                |
                |     m------k
                |     |        |
                |     |      |
                |     |      |
                |     i------j
                |
                 ---------------------->s

      organization in 2D for four points
      the desired point has the coordinates at(s,t)
*/

      /* height=C1+C2*s+C3*t+C4*s*t */
      C1 = field[i + j*noColumns];
            /* height at i */
      C2 = (field[(i+1) + j*noColumns]-field[i + j*noColumns])/(2*DIM);
            /* height at j minus height at i */
      C3 = (field[i + (j+1)*noColumns]-field[i + j*noColumns])/(2*DIM);
            /* height at m minus height at i */
```

```
      C4 = (field[i + j*noColumns]-field[(i+1) + j*noColumns]+
            field[(i+1) + (j+1)*noColumns]-field[i + (j+1)*noColumns])
(4*DIM*DIM);

      specifiedHeight=C1+C2*s+C3*t+C4*s*t;

      return(specifiedHeight);
}


/* calculates the grid points surrounding any given point */
void corners(double x, double y, int *iWanted, int *jWanted)
{
      double i, j;
      for(i=0.0 ; i<noRows-1 ; i+=1.0) {
            for(j=0.0 ; j<noColumns-1 ; j+=1.0) {
                  if( ((x+TOL)>=i*DIM) && ((x-TOL)<(i+1)*DIM) &&
((y+TOL)>=j*DIM) && ((y-TOL)<(j+1)*DIM) ){
                        *iWanted=(int)i;
                        *jWanted=(int)j;
                        /* printf("iWanted=%d, jWanted=%d\n", *iWanted,
*jWanted); */
                  }
            }
      }
      if (*iWanted < 0)
            *iWanted = 0 ;
      else if (*iWanted > noRows-1)
            *iWanted = noRows-1 ;

      if (*jWanted < 0)
            *jWanted = 0 ;
      else if (*jWanted > noColumns-1)
            *jWanted = noColumns-1 ;
}

/* calculates the pitch and roll for a given direction */
void pAndR(double *field,double x, double y, double *pitch, double
*roll, double *length)
{
      double zDifX, zDifY;
      int m,n, dxm, dxn, dym, dyn;

      corners(x,y,&m,&n);
      corners(x+deltaX,y,&dxm,&dxn) ;
      corners(x,y+deltaY,&dym,&dyn) ;
      zDifY=(heightMesh(field,dym,dyn,x,(y+deltaY))-
heightMesh(field,m,n,x,y));
      zDifX=(heightMesh(field,dxm,dxn,(x+deltaX),y)-
heightMesh(field,m,n,x,y));
      *pitch=(atan2(zDifY,deltaY))*R2D;
      *roll=(atan2(zDifX,deltaX))*R2D;
}

/* writes the virtual obstacles to the "map_data.txt" file for analysis
*/
```

```
void recordObs(double x[], double y[], int numberObstacles, double
*field)
{
      int i;
      double var = 2,coord1X, coord1Y, coord2X, coord2Y, coord3X,
coord3Y, coord4X, coord4Y;

      FILE *fpVirtual, *fpMatlab;

      if ((fpVirtual=fopen("map_data.txt","w"))==NULL){
            printf("Can't open 'map_data.txt' file \n");
            /* in case that there is no file exit */
            exit(1);
      }

      if ((fpMatlab = fopen("pObs.txt","w"))==NULL){
            printf("Can't open 'pObs.txt' file \n");
            /* in case that there is no file exit */
            exit(1);
      }

      fprintf(fpVirtual,"# this file contains virtual obstacles based
on bad slopes\n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"m %d\n", numberObstacles);
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"b 4\n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual," \n");

      for(i=0; i<numberObstacles; i++)
            fprintf(fpVirtual,"n 0 0 1 2 3 0.200000 4 5 6 4
0.000000\n");

      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"N\n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"#vertex data\n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"s 0 0 0.0 -0.707\n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"S\n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"g 999 999\n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"G\n");
      fprintf(fpVirtual," \n");

      /* define a polygon around the point of interest

                                  y
                       4*<---->|     *1
                    var  |
                                  |
                                  X------------->x
```

```
                        3*                  *2
      */

      for(i=0; i<numberObstacles; i++){
            coord1X = x[i] + var;
            coord1Y = y[i] + var;/* was y[i] */

            coord2X = x[i] + var;/* was x[i] */ /* was coordinate 2 and
+ */
            coord2Y = y[i] - var;

            coord3X = x[i] - var;
            coord3Y = y[i] - var;/* was y[i] */

            coord4X = x[i] - var;/* was x[i] */
            coord4Y = y[i] + var;

/*          checkHeights(coord1X, coord1Y, coord2X, coord2Y, coord3X,
coord3Y, coord4X, coord4Y,
                  field);
*/
            fprintf(fpVirtual,"p\n");
            fprintf(fpVirtual,"%lf %lf\n",coord1X, coord1Y);
            fprintf(fpVirtual,"%lf %lf\n",coord2X, coord2Y);
            fprintf(fpVirtual,"%lf %lf\n",coord3X, coord3Y);
            fprintf(fpVirtual,"%lf %lf\n",coord4X, coord4Y);
            fprintf(fpVirtual," \n");

            fprintf(fpMatlab,"%lf %lf\n",coord1X, coord1Y);
            fprintf(fpMatlab,"%lf %lf\n",coord2X, coord2Y);
            fprintf(fpMatlab,"%lf %lf\n",coord3X, coord3Y);
            fprintf(fpMatlab,"%lf %lf\n",coord4X, coord4Y);

      }

      fprintf(fpVirtual,"P\n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"q\n");
      fprintf(fpVirtual,"-1000 -1000\n");
      fprintf(fpVirtual,"-1000 1000\n");
      fprintf(fpVirtual,"1000 1000\n");
      fprintf(fpVirtual,"1000 -1000\n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual," \n");
      fprintf(fpVirtual,"*\n");
      fprintf(fpVirtual,"*");

printf("closing the file\n");
      fclose(fpMatlab);
      fclose(fpVirtual);
}

void checkHeights(double c1x, double c1y, double c2x, double c2y,
                        double c3x, double c3y, double c4x, double
c4y,
                        double *field)
```

```
{
	int m1, m2, m3, m4, n1, n2, n3, n4;
	double theta1, theta2, theta3, theta4;
	double heightDif1, heightDif2, heightDif3, heightDif4;

	corners(c1x,c1y,&m1,&n1);
	corners(c2x,c2y,&m2,&n2);
	corners(c3x,c3y,&m3,&n3);
	corners(c4x,c4y,&m4,&n4);

	heightDif1 = heightMesh(field,m1,n1,c1x,c1y) -
heightMesh(field,m2,n2,c2x,c2y);
	heightDif2 = heightMesh(field,m2,n2,c2x,c2y) -
heightMesh(field,m3,n3,c3x,c3y);
	heightDif3 = heightMesh(field,m3,n3,c3x,c3y) -
heightMesh(field,m4,n4,c4x,c4y);
	heightDif4 = heightMesh(field,m4,n4,c4x,c4y) -
heightMesh(field,m1,n1,c1x,c1y);

	theta1 = atan2(heightDif1, (c1y - c2y))*R2D;
	theta2 = atan2(heightDif2, (c2x - c3x))*R2D;
	theta3 = atan2(heightDif3, (c3y - c4y))*R2D;
	theta4 = atan2(heightDif4, (c4x - c1x))*R2D;

	if(fabs(theta1) > CRITICAL_SLOPE)
		printf("bad theta 1 %lf\n", theta1);

	if(fabs(theta2) > CRITICAL_SLOPE)
		printf("bad theta 2 %lf\n", theta2);

	if(fabs(theta3) > CRITICAL_SLOPE)
		printf("bad theta 3 %lf\n", theta3);

	if(fabs(theta4) > CRITICAL_SLOPE)
		printf("bad theta 4 %lf\n", theta4);
scanf("%c");
	return(0);
}

void extractRealData(int x)
{
	char ch;
	int realPoints = 0, i;
	double latitude, longitude, altitude, garbage;
	double garbage1, garbage2, garbage3, garbage4, garbage5,
garbage6, garbage7, garbage8;
	double garbage9, garbage10, garbage11, garbage12, garbage13,
garbage14, garbage15;
	double minX = 99999, maxX = -1, minY = 99999, maxY = -1;
	double firstPointLat = 29.6462735, firstPointLon = -82.3536036,
north, east;
	double terrainNorth, terrainEast;

	FILE *fpReal, *fpTerrain;

	if ((fpReal = fopen("Vladmap.dat","r")) == NULL){
		printf("Can't open 'Vladmap.dat' file \n");
```

```
            /* in case that there is no file exit */
            exit(1);
      }

      if ((fpTerrain = fopen("terrain.txt","w")) == NULL){
            printf("Can't open 'terrain.txt' file \n");
            /* in case that there is no file exit */
            exit(1);
      }
      while(ch=getc(fpReal) != '*'){
            fscanf(fpReal,"%lf,%lf,%lf,%lf, %lf, %lf, %lf, %lf, %lf,
%lf, %lf, %lf, %lf, %lf, %lf, %lf, %lf, %lf, %lf, %lf\n",
                  &garbage, &latitude, &longitude, &altitude,
                  &garbage1, &garbage2, &garbage3, &garbage4,
&garbage5, &garbage6, &garbage7, &garbage8,
                  &garbage9, &garbage10, &garbage11, &garbage12,
&garbage13, &garbage14, &garbage15);

            /* find minimum and maximum for x and y */
/*          if(latitude > maxX)
                  maxX = latitude;
            if(latitude < minX)
                  minX = latitude;
            if(longitude > maxY)
                  maxY = longitude;
            if(longitude < minY)
                  minY = longitude;
*/
            /* convert to a local NED coord system */
            terrainNorth = 0.0;
            terrainEast = 0.0;

            north = (latitude - firstPointLat) * D2R * EARTH_RADIUS ;
            east = (longitude - firstPointLon) * D2R * EARTH_RADIUS *
cos(firstPointLat*D2R) ;

            if(north > maxX)
                  maxX = north;
            if(north < minX)
                  minX = north;
            if(east > maxY)
                  maxY = east;
            if(east < minY)
                  minY = east;

            fprintf(fpTerrain,"%lf %lf %lf\n", north, east, altitude);
                  realPoints++;
      }

      fprintf(fpTerrain,"*\n");
      fclose(fpReal);

      fclose(fpTerrain);

      printf("minX = %lf, maxX = %lf, minY = %lf, maxY = %lf\n", minX,
maxX, minY, maxY);
```

```
    return(0);
}
```

```
/*  obstacles.cpp
  Vlad Hurezeanu

  This program creates the path for an autonomous vehicle in a 3D
environment
  for the purpose of surveying the respective terrain.
  The coordinates of the corners of the terrain are introduced in any
order.
  The terrain can have any shape.
////////////////////////////////////////////////////////////////////
///////////*/

#include "includes.h"
#include "fieldSweepLib.h"
#include "ufMathLib.h"

#ifndef EARTH_RADIUS
#define EARTH_RADIUS  6.378137e6
#endif

#ifndef M_PI
#define M_PI 3.14159265359
#endif

#define INFINITY      1e6
#define TOL           1e-3
#define TRUE          1
#define FALSE         0
#define INCREASING    1
#define DECREASING    -1
#define DBS           1.0
#define D2R           (M_PI/180)
#define R2D           (180/M_PI)
#define SIZE              32
#define DIM           5

#define POLYGON_DEFINED_CORRECTLY  0
#define POLYGON_DEFINED_CCW        1
#define POLYGON_INTERSECTS_ITSELF  2

#define NORTH 0
#define EAST  1
#define LAT   0
#define LON   1

/* define grid size */
//#define noColumns                44
//#define noRows                   69
#define noColumns       32
#define noRows                32

struct _wayPoints {
      double x ;
      double y ;
      int rowNumber ;
      int lineSegment ;
      struct _wayPoints *next ;
```

```
      struct _wayPoints *prev ;
} ;

struct line
{
  double m;
  double b;
};

struct line_segment
{
  double x1;
  double y1;
  double x2;
  double y2;
};

/* geoBoundryPts was not changed with boundryPts */
double fieldSweep(double sweepAngle, double *field, double d, int
numpoints,
                          double geoBoundryPts[][2],float rowDist,float
minTurnRad,
                          struct _geoPoint **plan, double *costSum, int
value)
{
      /* variable declarations */
      int i, k, current_pos, num_rows, numRowsOrig, numWayPoints,
*slope, count;
      int wantedRowNumber, wantedLineSegment;
      double *dist,theta,x,y,*field_coord_x,*field_coord_y;
      double virtualObstacleX[500], virtualObstacleY[500];
      double a, b ,longest_dist=0,start1_x,start1_y,start2_x,start2_y;
      int secondCount=0, odd = FALSE;
      char done;
      struct line l ;
      struct line_segment *ls;
      double *boundryPts[2] ;
      struct _wayPoints *wayPoints=NULL, *tmp;
      struct _wayPoints *orderedWayPoints=NULL, *tmp1,*tmpEnd ;
      struct _wayPoints *cur, *next ;
      struct _geoPoint *tmpGeo ;
      FILE *fp, *fp2, *fpm, *fp5, *fp6;
      FILE *fpMinPts, *fpmLocal;

      double length, xNext, yNext, firstRow;
      int m, n, current;
      double cd, tmpAng, tmpCost, z, h;

      /* function prototypes */
      int intersectionPoints(struct line,struct line_segment,double
*,double *);
      int getNextWayPoint(struct _wayPoints **wayPoints, int row, int
direction,
                  struct _wayPoints **nextWayPoint);
      int getConcavePoints(struct _wayPoints **wayPoints, int, double
[], double [], int);
```

```
      int polygonErrorChecking(double xpoly[], double ypoly[], int
num_vertices) ;
      int avoidObstacles(struct _wayPoints **orderedWayPoints, double
[], double [],
                         float, int, int, double, int);
      int smoothWayPoints(float, int, struct
_wayPoints**orderedWayPoints);
      double cost(double *field, double x, double y, double d, int
value, int *unknown);
      double heightMesh(double *field, int i, int j, double s, double
t);
      void corners(double x, double y, int *iWanted, int *jWanted);
      void recordObs(double x[], double y[], int numberObstacles);

      if (numpoints < 3) {
            printf("requires 3 or more boundry points\r\n") ;
            return(-1) ;
      }

      /* allocate memory for variables */
      if ((field_coord_x = (double *)(malloc(sizeof(double) *
(numpoints+1))))
      == NULL){
            printf("error allocating memory!!!\n");
            return(-1);
      }

      if ((field_coord_y = (double *)(malloc(sizeof(double) *
(numpoints+1))))
            == NULL){
            free(field_coord_x);
            printf("error allocating memory!!!\n");
            return(-1);
      }

      if ((dist = (double *)(malloc(sizeof(double) * (numpoints+1))))
            == NULL){
            free(field_coord_x);
            free(field_coord_y);
            printf("error allocating memory!!!\n");
            return(-1);
      }

      if ((ls = (struct line_segment *)(malloc(sizeof(struct
line_segment) * (numpoints+1))))
            == NULL){
            free(field_coord_x);
            free(field_coord_y);
            free(dist);
            printf("error allocating memory!!!\n");
            return(-1);
      }

      if ((boundryPts[0] = (double *)(malloc(sizeof(double) *
(numpoints))))
            == NULL){
            free(field_coord_x);
```

```
            free(field_coord_y);
            free(dist);
            free(ls);
            printf("error allocating memory!!!\n");
            return(-1);
      }

      if ((boundryPts[1] = (double *)(malloc(sizeof(double) *
(numpoints))))
            == NULL){
            free(field_coord_x);
            free(field_coord_y);
            free(dist);
            free(ls);
            free(boundryPts[0]);
            printf("error allocating memory!!!\n");
            return(-1);
      }

      /* delete current plan if it exists */
      while ((*plan) != NULL) {
            tmpGeo = (*plan) ;
            *plan = tmpGeo->next ;
            free(tmpGeo) ;
      }

      /* convert to a local NED coord system */
      boundryPts[NORTH][0] = 0.0;
      boundryPts[EAST][0] = 0.0;
      for (i=1; i<numpoints; i++) {
            boundryPts[NORTH][i] = (geoBoundryPts[i][LAT] -
geoBoundryPts[0][LAT]) *
                  D2R * EARTH_RADIUS ;
            boundryPts[EAST][i]  = (geoBoundryPts[i][LON] -
geoBoundryPts[0][LON]) *
                  D2R * EARTH_RADIUS * cos(geoBoundryPts[0][LAT]*D2R) ;
      }
      /* calculate distance of each boundry line segment */
      for(i=0;i<(numpoints-1);++i)
      {
            dist[i]=sqrt(pow((boundryPts[NORTH][i+1]-
boundryPts[NORTH][i]),2)
                  +pow((boundryPts[EAST][i+1]-boundryPts[EAST][i]),2));
      }
      dist[(numpoints-1)]=sqrt(pow((boundryPts[NORTH][0]-
boundryPts[NORTH][(numpoints-1)]),2)
            +pow((boundryPts[EAST][0]-boundryPts[EAST][(numpoints-
1)]),2));

      /* find longest boundry line segment */
      longest_dist=dist[0];
      current_pos = 0 ;
      for(i=1;i<numpoints;++i)
      {
                  if (dist[i]>longest_dist)
                  {
                        longest_dist=dist[i];
```

```
                        current_pos=i;
                    }

        }
        start1_x=boundryPts[NORTH][current_pos];
        start1_y=boundryPts[EAST][current_pos];

        if(current_pos==(numpoints-1))
        {
              start2_x=boundryPts[NORTH][0];
              start2_y=boundryPts[EAST][0];
        }
        else
        {
              start2_x=boundryPts[NORTH][current_pos+1];
              start2_y=boundryPts[EAST][current_pos+1];
        }

        /* convert to local XYZ coord system */
        /* theta=atan2((start2_y-start1_y),(start2_x-start1_x)); */
        theta = sweepAngle;
        for(i=0;i<numpoints;++i)
        {
              if ((i+current_pos) < numpoints) {

        field_coord_x[i]=(boundryPts[NORTH][i+current_pos])*(cos(theta))
                        +(boundryPts[EAST][i+current_pos])*(sin(theta))
                        -(boundryPts[NORTH][current_pos]*cos(theta)
                        +boundryPts[EAST][current_pos]*sin(theta));
                    field_coord_y[i]=-
(boundryPts[NORTH][i+current_pos])*(sin(theta))
                        +(boundryPts[EAST][i+current_pos])*(cos(theta))
                        -(-boundryPts[NORTH][current_pos]*sin(theta)
                        +boundryPts[EAST][current_pos]*cos(theta));
              }
              else /* ((i+current_pos)>=numpoints) */ {
                    field_coord_x[i]=(boundryPts[NORTH][i+current_pos-
(numpoints)])*(cos(theta))
                        +(boundryPts[EAST][i+current_pos-
(numpoints)])*(sin(theta))
                        -(boundryPts[NORTH][current_pos]*cos(theta)
                        +boundryPts[EAST][current_pos]*sin(theta));
                    field_coord_y[i]=-(boundryPts[NORTH][i+current_pos-
(numpoints)])*(sin(theta))
                        +(boundryPts[EAST][i+current_pos-
(numpoints)])*(cos(theta))
                        -(-boundryPts[NORTH][current_pos]*sin(theta)
                        +boundryPts[EAST][current_pos]*cos(theta));
              }
        }
        field_coord_x[numpoints]=field_coord_x[0];
        field_coord_y[numpoints]=field_coord_y[0];

        /* make sure boundry is define clockwise */
        if ((polygonErrorChecking(field_coord_x,field_coord_y,numpoints))
==
              POLYGON_INTERSECTS_ITSELF) {
```

```
            free(field_coord_x) ;
            free(field_coord_y) ;
            free(dist) ;
            free(ls) ;
            free(boundryPts[0]) ;
            free(boundryPts[1]) ;
            printf("Error in boundry points!!!\r\n") ;
            return(-1) ;
      }

      for (i=0;i<numpoints;i++)
      {
            ls[i].x1=field_coord_x[i];
            ls[i].y1=field_coord_y[i];
            ls[i].x2=field_coord_x[i+1];
            ls[i].y2=field_coord_y[i+1];
      }

      /* go down the y-axis until no more intersection points */
      done=FALSE ;
      firstRow = 0.0 ;
      while(!done) {
            done=TRUE;
            firstRow -= rowDist ;
            for(i=0;i<numpoints;i++) {
                  l.m=0;
                  l.b=firstRow ;
                  if((intersectionPoints(l,ls[i],&x,&y))==TRUE)
                        done=FALSE;
            }
      }
      firstRow += rowDist ;

      /* find all intersection points that will make up the path */
      num_rows = 0 ;
      done=FALSE;
      while(!done)
      {
            done=TRUE;
            for(i=0;i<numpoints;i++)
            {
                  l.m=0;
                  l.b=firstRow+rowDist*num_rows;
                  if((intersectionPoints(l,ls[i],&x,&y))==TRUE)
                  {
                        done=FALSE;
                        if (wayPoints == NULL) {
                              wayPoints = (struct
_wayPoints*)malloc(sizeof(struct _wayPoints)) ;
                              if (wayPoints == NULL){
                                    free(field_coord_x);
                                    free(field_coord_y);
                                    free(dist);
                                    free(ls);
                                    free(boundryPts[0]);
                                    free(boundryPts[1]);
```

```
                                    printf("error allocating
memory!!!\n");
                                    return(-1);
                              }
                              wayPoints->x = x ;
                              wayPoints->y = y ;
                              wayPoints->rowNumber = num_rows+1 ;
                              wayPoints->lineSegment = i ;
                              wayPoints->next = NULL ;
                        }
                        else {
                              tmp = wayPoints ;
                              while (tmp->next != NULL)
                                    tmp = tmp->next ;
                              tmp->next = (struct
_wayPoints*)malloc(sizeof(struct _wayPoints)) ;
                              if(tmp->next == NULL){
                                    free(field_coord_x);
                                    free(field_coord_y);
                                    free(dist);
                                    free(ls);
                                    free(boundryPts[0]);
                                    free(boundryPts[1]);
                                    while(wayPoints != NULL){
                                          tmp = wayPoints;
                                          wayPoints = wayPoints->next;
                                          free(tmp);
                                    }
                                    printf("error allocating
memory!!!\n");
                                    return(-1);
                              }
                              tmp = tmp->next ;
                              tmp->x = x ;
                              tmp->y = y ;
                              tmp->rowNumber = num_rows+1 ;
                              tmp->lineSegment = i ;
                              tmp->next = NULL ;
                        }
                  }
            }
            if (done == FALSE)
                  num_rows++ ;

      }
      numRowsOrig=num_rows;
      /* check if odd number of rows */
      if (num_rows % 2) {
            /* make number of rows even */
            num_rows += 1 ;

            /* establish the value of 'odd' to be used in the obstacle
avoidance */
            odd = TRUE;

            /* find end of way point list */
            tmpEnd = wayPoints ;
```

```
            while (tmpEnd->next != NULL) {
                  if (tmpEnd->rowNumber >= num_rows/2)
                        tmpEnd->rowNumber+=1 ;
                  tmpEnd = tmpEnd->next ;
            }
            /* check last point */
            if (tmpEnd->rowNumber >= num_rows/2)
                  tmpEnd->rowNumber+=1 ;

            /* add center row twice */
            tmp = wayPoints ;
            while(tmp != NULL) {
                  if (tmp->rowNumber == num_rows/2+1) {
                        tmpEnd->next = (struct _wayPoints
*)malloc(sizeof(struct _wayPoints)) ;
                        if(tmpEnd == NULL){
                              free(field_coord_x);
                              free(field_coord_y);
                              free(dist);
                              free(ls);
                              free(boundryPts[0]);
                              free(boundryPts[1]);
                              while(wayPoints != NULL){
                                    tmp = wayPoints;
                                    wayPoints = wayPoints->next;
                                    free(tmp);
                              }
                              printf("error allocating memory!!!\n");
                              return(-1);
                        }
                        tmpEnd = tmpEnd->next ;
                        tmpEnd->x = tmp->x ;
                        tmpEnd->y = tmp->y ;
                        tmpEnd->rowNumber = num_rows/2 ;
                        tmpEnd->lineSegment = tmp->lineSegment ;
                        tmpEnd->next = NULL ;
                  }
                  tmp = tmp->next ;
            }
      }

      /* put points in order */
      orderedWayPoints = NULL ;

      k = num_rows / 2 ;

      for (i=0; i<k; i++) {
            while (getNextWayPoint(&wayPoints,i+1,INCREASING,&tmp1) ==
0) {
                  if (orderedWayPoints == NULL) {
                        orderedWayPoints = tmp1 ;
                        tmp = orderedWayPoints ;
                  }
                  else {
                        tmp->next = tmp1 ;
                        tmp = tmp->next ;
                  }
```

```
            }

            while (getNextWayPoint(&wayPoints, k+i+1, DECREASING,
&tmp1) == 0) {
                  if (orderedWayPoints == NULL) {
                        orderedWayPoints = tmp1 ;
                        tmp = orderedWayPoints ;
                  }
                  else {
                        tmp->next = tmp1 ;
                        tmp = tmp->next ;
                  }
            }
      }
      tmp->next = NULL ;  /* end of list */

      /* if coordinate system is rotated boundary points are added */
      cur=orderedWayPoints->next;
      while((cur != NULL) && (cur->next != NULL)){
            next = cur->next;
            if((cur->rowNumber == next->rowNumber)
                  && (next->next != NULL)){
                  cur = next;
                  next = next->next;
            }
            wantedLineSegment = cur->lineSegment;
            wantedRowNumber = cur->rowNumber;
            if(cur->lineSegment != next->lineSegment){
                  cur->next = (struct _wayPoints*)malloc(sizeof(struct
_wayPoints)) ;
                  cur=cur->next ;
                  cur->next=next;
                  cur->x = ls[wantedLineSegment].x1 ;
                  cur->y = ls[wantedLineSegment].y1 ;
                  cur->rowNumber = wantedRowNumber;
                  cur->lineSegment = wantedLineSegment ;
            }
            cur=next->next;
      }

      /* outputs field in the local coordinate system */
      fp5 = fopen("world_order5.txt","w");
      tmp = orderedWayPoints ;
      numWayPoints=0;
            while (tmp != NULL) {
                  fprintf(fp5,"o %lf %lf %d\n",tmp->x, tmp->y, tmp-
>rowNumber) ;
                  tmp = tmp->next ;
      }
      fprintf(fp5,"*\n") ;
      fclose(fp5);

      /* check for concave field and insert boundry pts if needed */
      /*
getConcavePoints(&orderedWayPoints,num_rows,field_coord_x,field_coord_y
,numpoints) ; */
```

```
	/* check the images */
	fpmLocal = fopen("pLocal.m","w");
	tmp = orderedWayPoints ;
	fprintf(fpmLocal,"A=[");
		while (1) {
			fprintf(fpmLocal,"%lf,",tmp->x) ;
			if(tmp->next == NULL)
				break;
			tmp = tmp->next ;
	}
	fprintf(fpmLocal,"]\n");

	tmp = orderedWayPoints ;
	fprintf(fpmLocal,"B=[");
		while (tmp != NULL) {
			fprintf(fpmLocal,"%lf,",tmp->y) ;
			tmp = tmp->next ;
	}
	fprintf(fpmLocal,"]\n");
	fprintf(fpmLocal,"plot(A, B)");
	fclose(fpmLocal);

	/* convert back to NED coord sys */
	tmp = orderedWayPoints ;
	numWayPoints = 0 ;
	while (tmp != NULL){
		numWayPoints+=1 ;
		a=tmp->x*cos(theta)-tmp-
>y*sin(theta)+boundryPts[NORTH][current_pos];
		b=tmp->x*sin(theta)+tmp-
>y*cos(theta)+boundryPts[EAST][current_pos];
		tmp->x = a;
		tmp->y = b;
		tmp = tmp->next;
	}

	/* displays an array of information about the 3D field */
	fpMinPts = fopen("allPoints.txt","w");

	/* find the cost for the path */
	cur = orderedWayPoints ;
	*costSum = 0.0 ;
	h = 0.0;
	count = 0;
	while((cur != NULL)&&(cur->next != NULL)){
		length=sqrt(pow(cur->next->x - cur->x, 2)+pow(cur->next->y
- cur->y,2));
		if(d>=length){
			/* in case d is bigger than the length between 2
points */
			cost(field, cur->x, cur->y, d, value, &slope);
		}
		else {
			for(cd=0.0 ; cd<length ; cd+=d){
				tmpAng = atan2(cur->next->y - cur->y, cur-
>next->x - cur->x) ;
```

```
                              /* tmpAng = atan2(cur->next->x - cur->x,
cur->next->y - cur->y) ; */
                              x = cur->x + cd*cos(tmpAng) ;
                              y = cur->y + cd*sin(tmpAng) ;
                              xNext = 0.0 ; /* not used */
                              yNext = 0.0 ; /* not used */
                              if (value ==0)
                                    slope = 1;
                              tmpCost = cost(field, x, y, d, value,
&slope);
                              *costSum += tmpCost ;
                              if (value == 0){
                                    if (slope == 0){
                                          /* record the bad slopes when
the path is minimum */
                                          virtualObstacleX[count] = x;
                                          virtualObstacleY[count] = y;
                                          printf("x=%lf, y=%lf\n",
x,y);
                                          count++;
                                    }
                              }
                              /* find the corners of the cell in order
to find height 'z' */
                              corners(x,y,&m,&n);
                              z = heightMesh(field,m,n,x,y);
                              /*if(h<z){
                                    h=z;
                                    xNext = x;
                                    yNext = y;
                              }*/
                              fprintf(fpMinPts,"%lf %lf %d %d %lf %lf
%lf\n", x, y, m, n, z, tmpCost, *costSum);
                              /* printf("current cost = %lf, total cost
= %lf\n",tmpCost,*costSum); */
                              /* printf("x = %lf, y = %lf, ang =
%lf\n", x, y, tmpAng*RTD); */
                        }
                  }
                  cur = cur->next;
            }
            fclose(fpMinPts);

            /* eliminate the same obstacles */
            if(value == 0){
                  current = 0;
                  while(current <= count){
                        for(i=0; i<(current); i++){
                              if(((virtualObstacleX[current]+TOL) >=
virtualObstacleX[i])&&
                                    ((virtualObstacleX[current]-TOL) <=
virtualObstacleX[i])&&
                                    ((virtualObstacleY[current]+TOL) >=
virtualObstacleY[i])&&
                                    ((virtualObstacleY[current]-TOL) <=
virtualObstacleY[i])){
                                    secondCount++;
```

```
                        }
                  }
                  current++;
            }
      }
printf("original is %d, and repeated is %d\n", count,secondCount);
      /* substract the repeated obstacles form the total number */
      count = count - secondCount;

      /* write to the file "map_data.txt" the obstacles */
      if (value == 0)
            recordObs(virtualObstacleX, virtualObstacleY, count,
field);

      /* convert back to XYZ coord sys */
      tmp = orderedWayPoints ;
      while (tmp != NULL){
            a=tmp->x*cos(theta)+tmp->y*sin(theta)-
boundryPts[NORTH][current_pos]*cos(theta)+
                  boundryPts[EAST][current_pos]*sin(theta);
            b=-tmp->x*sin(theta)+tmp-
>y*cos(theta)+boundryPts[NORTH][current_pos]*sin(theta)-
                  boundryPts[EAST][current_pos]*cos(theta);
            tmp->x = a;
            tmp->y = b;
            tmp = tmp->next;
      }

      /* check for obstacles when the best path is found */
      if (value == 0){
            if (count != 0){
                  printf("checking for obstacles\n");
                  avoidObstacles(&orderedWayPoints,field_coord_x,
field_coord_y, rowDist,
                                    numpoints, numRowsOrig, sweepAngle,
odd);
            }
            else
                  printf("\n.............!!!   There are no
obstacles   !!!.............\n\n");
      }

      /* displays the field map in local coordinate system including
obstacles */
      fp6 = fopen("world_order6.txt","w");
      tmp = orderedWayPoints ;
            while (tmp != NULL) {
                  fprintf(fp6,"%lf %lf %d\n",tmp->x, tmp->y, tmp-
>rowNumber) ;
                  tmp = tmp->next ;
      }
      fclose(fp6);

      /* convert back to NED coord sys */
      tmp = orderedWayPoints ;
      numWayPoints=0;
      while (tmp != NULL){
```

```
            numWayPoints+=1;
            a=tmp->x*cos(theta)-tmp-
>y*sin(theta)+boundryPts[NORTH][current_pos];
            b=tmp->x*sin(theta)+tmp-
>y*cos(theta)+boundryPts[EAST][current_pos];
            tmp->x = a;
            tmp->y = b;
            tmp = tmp->next;
      }

      /* accounts for the turning radius of the vehicle */
      if (value == 0){
            printf("the number of way points is %d\n", numWayPoints);
            smoothWayPoints(minTurnRad,numWayPoints,&orderedWayPoints);
      }

      /* displays information about the path to be used for OpenGl
program */
      fp = fopen("world_order.txt","w");
      tmp = orderedWayPoints ;
      numWayPoints=0;
            while (tmp != NULL) {
                  numWayPoints+=1;
                  corners(tmp->x,tmp->y,&m,&n);
                  z = heightMesh(field,m,n,tmp->x,tmp->y);
                  //fprintf(fp,"o %lf %lf %d\n",tmp->x, tmp->y, tmp-
>rowNumber) ;
                  fprintf(fp,"o %lf %lf %lf\n",tmp->x, tmp->y, z) ;
                  tmp = tmp->next ;
      }
      fprintf(fp,"*\n") ;
      fclose(fp);

      /* displays the same as above but to be used with Matlab */
      fp2 = fopen("world_order2.txt","w");
      tmp = orderedWayPoints ;
      numWayPoints=0;
            while (tmp != NULL) {
                  corners(tmp->x,tmp->y,&m,&n);
                  z = heightMesh(field,m,n,tmp->x,tmp->y);
                  /* fprintf(fp,"o %lf %lf %d\n",tmp->x, tmp->y, tmp-
>rowNumber) ; */
                  fprintf(fp2,"%lf %lf %lf\n",tmp->x, tmp->y, z) ;
                  tmp = tmp->next ;
      }
      fclose(fp2);

      fpm = fopen("points.m","w");
      tmp = orderedWayPoints ;
      fprintf(fpm,"A=[");
            while (tmp != NULL) {
                  fprintf(fpm,"%lf,",tmp->x) ;
                  tmp = tmp->next ;
      }
      fprintf(fpm,"]\n");

      tmp = orderedWayPoints ;
```

```
      fprintf(fpm,"B=[");
            while (tmp != NULL) {
                  fprintf(fpm,"%lf,",tmp->y) ;
                  tmp = tmp->next ;
      }
      fprintf(fpm,"]\n");
      fprintf(fpm,"plot(A, B)");
      fclose(fpm);

      /* convert back to geodetic coord system */
      while(orderedWayPoints != NULL) {
            tmp = orderedWayPoints ;
            if ((*plan) == NULL) {
                  (*plan) = (struct _geoPoint *)(malloc(sizeof(struct
_geoPoint))) ;
                  tmpGeo = (*plan) ;
            }
            else {
                  tmpGeo->next = (struct _geoPoint
*)(malloc(sizeof(struct _geoPoint))) ;
                  tmpGeo = tmpGeo->next ;
            }
            tmpGeo->lat = geoBoundryPts[0][LAT] + (orderedWayPoints->x
/ EARTH_RADIUS) * R2D ;
        tmpGeo->lon = geoBoundryPts[0][LON] +
                  (orderedWayPoints->y / (EARTH_RADIUS *
cos(geoBoundryPts[0][LAT]*D2R))) * R2D ;
            tmpGeo->next = NULL ;

            orderedWayPoints = orderedWayPoints->next ;
            free(tmp) ;
      }

      /* free memory */
      free(field_coord_x) ;
      free(field_coord_y) ;
      free(dist) ;
      free(ls) ;
      free(boundryPts[0]) ;
      free(boundryPts[1]) ;

      return(0);
}

/* determines if an intersection exists for a line given the slope, m,
   and the y intercept,b, where:
        y = mx + b
   or,
        x = b, m = INFINITY
   and a line segment given two points:
        (x1,y1) and (x2,y2)
   return the intersection, (x,y), if it exists */


int intersectionPoints(struct line l, struct line_segment ls, double
*x, double *y)
{
```

```
      double m1,b1,m2,b2;
      double x_min, x_max, y_min, y_max;

      m1=l.m;  /* always zero */
      b1=l.b;

      /* x = constant */
      if (fabs(ls.x1-ls.x2) < TOL)
      {
            *x=ls.x1;
            *y=m1*ls.x1+b1;
      }
      /* y = constant */
      else if (fabs(ls.y1-ls.y2) < TOL)
            return(0);
    /* y = mx + b */
      else
      {
            m2=(ls.y2-ls.y1)/(ls.x2-ls.x1);
            b2=-m2*ls.x1+ls.y1;

            *x=(b1-b2)/(m2);
            *y=(m2*b1-m1*b2)/(m2);
      }

      /* determine if on line segment */
      if (ls.x1 < ls.x2)
      {
            x_min = ls.x1;
            x_max = ls.x2;
      }
      else
      {
            x_min = ls.x2;
            x_max = ls.x1;
      }
      if (ls.y1 < ls.y2)
      {
            y_min = ls.y1;
            y_max = ls.y2;
      }
      else
      {
            y_min = ls.y2;
            y_max = ls.y1;
      }

      if ((*x >= x_min-TOL) && (*y >= y_min-TOL) && (*x <= x_max+TOL)
&& (*y <= y_max+TOL))
            return(1);
      else
            return(0);
}

int getNextWayPoint(struct _wayPoints **wayPoints, int row, int
direction,
               struct _wayPoints **nextWayPoint)
```

```
{
  struct _wayPoints *prev, *cur, *next ;
  double xValue ;
  int numPts ;

  cur = *wayPoints ;

  /* check if there are any points in list */
  if (cur == NULL) {
    /* no way points */
    return(-1) ;
  }

  /* determine how many points have the  */
  /* correct row and the min/max x value */
  numPts = 0 ;
  while (cur != NULL) {
    if (cur->rowNumber == row) {
      if (numPts == 0)
        xValue = cur->x ;
      else {
        if (direction == INCREASING) {
          /* want to find minimum */
          if (cur->x < xValue)
            xValue = cur->x ;
        }
        else /* direction == DECREASING */ {
          /* want to find maximum */
          if (cur->x > xValue)
            xValue = cur->x ;
        }
      }
      numPts++ ;
    }
    cur = cur->next ;
  }

  if (numPts == 0) {
    /* no points in list with correct row number */
    return(-1) ;
  }
  else /* numPts != 0 */ {
    /* find element in list with correct */
    /* row number and xValue             */

    prev = NULL ;
    cur  = *wayPoints ;
    next = cur->next ;

    while (cur != NULL) {
      if ((cur->rowNumber == row) && (cur->x == xValue))
        break ;
      else {
        prev = cur ;
        cur  = next ;
        next = cur->next ;
      }
```

```
    }

      if (cur == NULL) {
      /* error:  could not find element */
      return(-1) ;
    }
    else {
      *nextWayPoint = cur ;

      /* take point out of list */
      if (prev != NULL) /* in the middle or at the end of the list */
        prev->next = next ;
      else
        *wayPoints = next ; /* at the beginning of the list */

      return(0) ;
    }
  }
}


int getConcavePoints(struct _wayPoints **orderedWayPoints,int num_rows,
double x[],
                              double y[], int numBoundryPts)
{
  struct _wayPoints *cur, *next ;
  int start, end ;

  /* check if there are any points in list */
  if (cur == NULL) {
    /* no way points */
    return(-1) ;
  }
      /* start at second point in list */
      cur = (*orderedWayPoints)->next;
      while((cur != NULL) && (cur->next != NULL)) {
            next = cur->next;
            if(cur->lineSegment!=next->lineSegment){
                  if(1 /*cur->rowNumber>num_rows/2*/){ /* INCREASING */
                        start=cur->lineSegment+1 ;   /* boundry pt to
start inserting */
                        end = next->lineSegment+1 ;  /* boundry pt to
end inserting */
                        if (start >= numBoundryPts)
                              start = 0 ;
                        if (end >= numBoundryPts)
                              end = 0 ;
                        while(start != end) {
                              cur->next = (struct _wayPoints
*)malloc(sizeof(struct _wayPoints)) ;
                              cur = cur->next ;
                              cur->x = x[start] ;
                              cur->y  = y[start] ;
                              cur->rowNumber = next->rowNumber ;
                              cur->lineSegment = -1 ;
                              start++;
                              if (start >= numBoundryPts)
```

```
                                    start = 0 ;
                              }
                        }
                        else{
                              start=cur->lineSegment ;   /* boundry pt to
start inserting */
                              end = next->lineSegment ;  /* boundry pt to end
inserting */
                              while(start != end) {
                                    cur->next = (struct _wayPoints
*)malloc(sizeof(struct _wayPoints)) ;
                                    cur = cur->next ;
                                    cur->x = x[start] ;
                                    cur->y  = y[start] ;
                                    cur->rowNumber = next->rowNumber ;
                                    cur->lineSegment = -1 ;
                                    start--;
                                    if (start <= 0)
                                          start = numBoundryPts - 1;

                              }
                        }
                        cur->next = next ;
                  }
            cur = next->next;
            }
return(0);
}

int avoidObstacles(struct _wayPoints **orderedWayPoints, double x[],
double y[], float rowDist,
                        int numBoundryPts, int numRows, double
theta, int oddEven)
{
      struct _wayPoints *cur, *next, *prev, *startRow, *endRow,
*nextStartRow;
      int lineNumber, segmentNumber, intNumber, start, end , time;

      struct line l ;
      struct line_segment ls[100], lsOrig[100];
      double a, b;// a for x, b for y for intersection points
      double firstRow, secondRow, middleRow, tempI=0, tempD=0, mid2;
      double obsX[100], obsY[100];
      int tempIseg=0, tempDseg=0, mid;
      int obstacle_int_ls[100], obstacleRow[100];

    struct map_struct *map  = 0;
    struct start_pose_struct * start_vertex = 0;
    struct goal_pose_struct *goal_vertex = 0;
    struct vehicle_struct *vehicle = 0;
    struct path_struct *path = 0;
      struct obst_struct *root = NULL;
      struct obst_struct *obs;
      struct vertex_struct *vertex;
      int i, j, numObs, value;

      FILE *fpMatlab;
```

```
      struct obst_struct *bdry_obs = 0;

      int leftorRight(int, int, int, double [], double [], int [],
struct line_segment [],
            double[],double[]);

      /* variables to be used for the expansion and self-intersecting
algorithms */
      map= create_map();
      start_vertex= (struct start_pose_struct *)malloc(sizeof(struct
start_pose_struct));
      start_vertex->next= (struct start_pose_struct
*)malloc(sizeof(struct start_pose_struct));
      goal_vertex= (struct goal_pose_struct *)malloc(sizeof(struct
goal_pose_struct));
      goal_vertex->next= (struct goal_pose_struct
*)malloc(sizeof(struct goal_pose_struct));
      vehicle=(struct vehicle_struct *)malloc(sizeof(struct
vehicle_struct));
      path = (struct path_struct *)calloc(1, sizeof(struct
path_struct));
      bdry_obs = (struct obst_struct *)malloc(sizeof(struct
obst_struct));

      printf("reading map...\n");
      /* reads map data from a data file into the structures in
structs.h */
    read_map_data(map, start_vertex, goal_vertex) ;

      printf("reading vehicle map...\n");
      /* fill in the vehicle structure and start and goal pose
structures */
      read_vehicle_data(vehicle);

      printf("preprocess\n");
      /* combines any intersecting obstacles into only one obstacle */
    preprocess(map, vehicle, start_vertex, goal_vertex, path);

      printf("defining obstacles...\n");

      /* defines the first, middle, and last row of any coordinate
system  */
/*    if((theta > 0.0 - TOL) && (theta < 0.0 + TOL)){
            printf("0 case\n");
            //firstRow = -(int)(SIZE*DIM*sin(theta)/rowDist)-1;
            firstRow = 0;
            //secondRow = (int)((noColumns*cos(theta)-
rowDist)/rowDist);
            secondRow = numRows;
      }
      else if((theta > 0.0 + TOL) && (theta < M_PI/2)){
            printf("bigger than 0\n");
            //firstRow = -(int)(SIZE*DIM*sin(theta)/rowDist);
            //secondRow = (int)(SIZE*DIM*cos(theta)/rowDist);
            firstRow = 0; // -3
            secondRow = numRows; //6
      }
```

```
      else if((theta > M_PI/2-TOL) && (theta < M_PI/2+TOL)){
            printf("90 case\n");
            firstRow = -(int)(SIZE*DIM*sin(theta)/rowDist);
            secondRow = (int)(SIZE*DIM*cos(theta)/rowDist);
      }
      else if((theta > M_PI/2) && (theta < M_PI-TOL)){
            printf("smaller than 180\n");
            firstRow = -(int)(SIZE*DIM*(sin(M_PI-theta)+cos(M_PI-
theta))/rowDist);
            secondRow = 0;
      }
      else if((theta > M_PI-TOL) && (theta < M_PI+TOL)){
            printf("180 case\n");
            firstRow = -(int)(SIZE*DIM*(sin(M_PI-theta)+cos(M_PI-
theta))/rowDist)-1;
            secondRow = 0;
      }
      middleRow = (firstRow)-(firstRow-secondRow)/2+1;
      middleRow = 4;
printf("first row is %lf\n", firstRow);
printf("second row is %lf\n", secondRow);
printf("middle row is %lf\n", middleRow);
*/
    root = map->obs;
      obs = map->obs;
printf("there are %d obstacles\n",map->num_obstacles);

      if ((fpMatlab = fopen("pObs2.txt","w"))==NULL){
            printf("Can't open 'pObs2.txt' file \n");
            /* in case that there is no file exit */
            exit(1);
      }

      for(numObs=0; numObs<(map->num_obstacles); ++numObs)  // was -2
      //for(numObs=0; numObs<(1); ++numObs)  // was -2
      {
            struct vertex_struct *vert_root;
            vert_root=map->obs->vertex;
            vertex=obs->vertex;
scanf("%c");
printf("OBSTACLE %d\n", numObs);
            /* assign vertices to a structure for intersection purposes
*/
            for(j=0; j<(obs->num_vertices); ++j)
            {
                  lsOrig[j].x1=vertex->x;
                  lsOrig[j].y1=vertex->y;
                  if (j == (obs->num_vertices-1)) {
                        lsOrig[j].x2 = lsOrig[0].x1 ;
                        lsOrig[j].y2 = lsOrig[0].y1 ;
                  }
                  else {
                        lsOrig[j].x2=vertex->next->x;
                        lsOrig[j].y2=vertex->next->y;
                  }
                  vertex= vertex->next;
            }
```

```
            /* convert obstacle coordinates to XYZ coordinates */
            for(i=0; i<(obs->num_vertices); i++){
                  ls[i].x1 =
((lsOrig[i].x1)*(cos(theta))+(lsOrig[i].y1)*(sin(theta)));
                  ls[i].y1 = (-
(lsOrig[i].x1)*(sin(theta))+(lsOrig[i].y1)*(cos(theta)));
                  ls[i].x2 =
((lsOrig[i].x2)*(cos(theta))+(lsOrig[i].y2)*(sin(theta)));
                  ls[i].y2 = (-
(lsOrig[i].x2)*(sin(theta))+(lsOrig[i].y2)*(cos(theta)));
                  fprintf(fpMatlab,"%lf, %lf\n", ls[i].x1, ls[i].y1);
            }

            cur = *orderedWayPoints;
            while((cur != NULL) && (cur->next != NULL))
            {
                  /* record current line number, segment number,
beggining and ending line coord */
                  lineNumber = cur->rowNumber;
                  segmentNumber = cur->lineSegment;
                  startRow = cur;
                  endRow = cur->next;

                  next = cur->next;

                  /* define the slope of the lines based on the number
of rows (odd/even) */
/*                  if (oddEven == 1){
                        if(lineNumber >= (int)((numRows+1)/2+1))
                              l.b = (firstRow + lineNumber -
2)*rowDist;
                        else
                              l.b = (firstRow + lineNumber -
1)*rowDist;
                  }
                  else
                        l.b = (firstRow + lineNumber - 1)*rowDist;
*/
                  if(startRow->y == endRow->y)
                        l.m = 0;
                  //else if(startRow->x == startRow->x)
                  //    l.m = 90;
                  else
                        l.m = (startRow->y - endRow->y)/(startRow->x -
endRow->x);

                  if((startRow->x == 0.0) && (startRow->y == 0.0))
                        l.b = 0.0;
                  else
                        l.b = (startRow->y * endRow->x - startRow->x *
endRow->y)/
                              (endRow->x - startRow->x);
//printf("lm = %lf\n", l.m);
printf("lb = %lf\n", l.b);
                  /* find intersection between each obstacle and all
the rows */
```

```
                  intNumber=0;
                  for(j=0;j<obs->num_vertices;j++)
                  {
                        if ((intersectionPoints(l, ls[j], &a, &b)) ==
TRUE) {
                              intNumber++;
                              obsX[intNumber-1]=a;
                              obsY[intNumber-1]=b;
//printf("obsX = %lf, obsY = %lf\n", obsX[intNumber-1], obsY[intNumber-
1]);
                              printf("x is %lf\n", x[numBoundryPts-1]);
                              if(obsX[intNumber-1] > x[numBoundryPts-
1]){
                                    obstacleRow[intNumber-1]=-1;
                                    intNumber --;
                              }
                              else if(obsX[intNumber-1] < x[0]){
                                    obstacleRow[intNumber-1]=-2;
                                    intNumber --;
                              }
                              else
                                    obstacleRow[intNumber-
1]=lineNumber;
//printf("obstcle row is %d\n", obstacleRow[intNumber-1]);
                              obstacle_int_ls[intNumber-1]=j;
                        }
                  }

                  /* arrange the intersection points in the correct
order */
                  for(i=0; i<=(intNumber-1); i=i+2){
                        /* check for ODD number of rows */
                        if(cur->rowNumber == (numRows+1)/2){
                              if(obsX[i]>obsX[i+1]){
                                    tempD = obsX[i];
                                    obsX[i] = obsX[i+1];
                                    obsX[i+1] = tempD;

                                    tempDseg = obstacle_int_ls[i];
                                    obstacle_int_ls[i] =
obstacle_int_ls[i+1];
                                    obstacle_int_ls[i+1] = tempDseg;
                              }

                              cur = startRow;
                              next = cur->next;

                              while((obsX[i] > cur->next->x) && (cur-
>rowNumber == cur->next->rowNumber)){
                                    cur = next;
                                    next = next->next;
                              }
                        }

                        /* INCREASING row numbers */
                        else if (cur->rowNumber > (numRows+1)/2){
                              if(obsX[i] < obsX[i+1]){
```

```
                                    tempI = obsX[i];
                                    obsX[i] = obsX[i+1];
                                    obsX[i+1] = tempI;

                                    tempIseg = obstacle_int_ls[i];
                                    obstacle_int_ls[i] =
obstacle_int_ls[i+1];
                                    obstacle_int_ls[i+1] = tempIseg;
                              }

                              cur = startRow;
                              next = cur->next;

                              while((obsX[i] < next->x) && (cur-
>rowNumber == next->rowNumber)){
                                    cur = next;
                                    next = cur->next;
                              }
                        }

                        /* DECREASING row numbers */
                        else if(cur->rowNumber < (numRows+1)/2){
                              if(obsX[i]>obsX[i+1]){
                                    tempD = obsX[i];
                                    obsX[i] = obsX[i+1];
                                    obsX[i+1] = tempD;

                                    tempDseg = obstacle_int_ls[i];
                                    obstacle_int_ls[i] =
obstacle_int_ls[i+1];
                                    obstacle_int_ls[i+1] = tempDseg;
                              }

                              cur = startRow;
                              next = cur->next;

                              while((obsX[i] > next->x) && (cur-
>rowNumber == next->rowNumber)){
                                    cur = next;
                                    next = cur->next;

                              }

                        }
                  }

                  /* add intersection points */
                  for (j=0; j<(intNumber-1); j=j+2) {
                        /* insert first intersection point */
                        cur->next=(struct _wayPoints
*)malloc(sizeof(struct _wayPoints));
                        cur=cur->next;
                        cur->x=obsX[j];
                        cur->y=obsY[j];
                        if(obstacleRow[j] == -1){
                              cur->rowNumber = -1;
                              printf("outside right\n");
```

```
                        }
                        else if(obstacleRow[j] == -2){
                              cur->rowNumber = -2;
                              printf("outside left\n");
                        }
                        else
                              cur->rowNumber = lineNumber ;
                        cur->lineSegment = segmentNumber ;
printf("the row number is %d\n", cur->rowNumber);
printf("numRows - 1 is %d\n", numRows-1);
                        /* decide left or right */
                        if(cur->rowNumber == 1)                    /* if
on the first row always CW */
                              value = TRUE;
                        else if(cur->rowNumber == (numRows+1)){   /* if
on the last row always CCW */
                              value = TRUE;
                              printf("last case\n");
                        }
                        else if(cur->rowNumber == -1)
                              value = FALSE;
                        else if(cur->rowNumber == -2)
                              value = TRUE;
                        else {
                              if(leftorRight(j, intNumber, (obs-
>num_vertices), obsX, obsY,
                                    obstacle_int_ls, ls,x,y))
                                    value = TRUE;
                              else
                                    value = FALSE;
                        }

                        if(value == TRUE)
                        {
                              /* add boundary points */
printf("CW directions\n");
                              start=obstacle_int_ls[j];   /* boundry pt
to start inserting */
                              end = obstacle_int_ls[j+1];  /* boundry
pt to end inserting */
                              if (start >= obs->num_vertices)
                                    start = 0 ;
                              if (end >= obs->num_vertices)
                                    end = 0 ;
                              while(start != end){
                                    /* next = cur->next ; */
                                    cur->next=(struct _wayPoints
*)malloc(sizeof(struct _wayPoints));
                                    cur=cur->next;
                                    cur->x = ls[start].x2;
                                    cur->y = ls[start].y2;
                                    cur->rowNumber = lineNumber ;
                                    cur->lineSegment = 9999 ;
                                    cur->next = next ;
                                    start++;time++;
                                    if (start >= obs->num_vertices)
                                          start = 0 ;
```

```
                                    }
                                /* add second intersection point */
                                cur->next=(struct _wayPoints
*)malloc(sizeof(struct _wayPoints));
                                cur=cur->next;
                                cur->x=obsX[j+1];
                                cur->y=obsY[j+1];
                                cur->rowNumber = lineNumber ;
                                cur->lineSegment = segmentNumber ;
                                cur->next = next ;
                          }
                          else{
                                /* add boundary points */
printf("CCW directions\n");
                                start=obstacle_int_ls[j];  /* boundry pt
to start inserting */
                                end = obstacle_int_ls[j+1];  /* boundry
pt to end inserting */
                                if (start >= obs->num_vertices)
                                      start = 0 ;
                                if (end >= obs->num_vertices)
                                      end = 0 ;
                                while(start != end){
                                      /* next = cur->next ; */
                                      cur->next=(struct _wayPoints
*)malloc(sizeof(struct _wayPoints));
                                      cur=cur->next;
                                      cur->x = ls[start].x1;
                                      cur->y = ls[start].y1;
                                      cur->rowNumber = lineNumber ;
                                      cur->lineSegment = segmentNumber ;
                                      cur->next = next ;
                                      start--;
                                      if (start < 0)
                                            start = (obs->num_vertices-1)
;
                                      }
                                /* add second intersection point */
                                cur->next=(struct _wayPoints
*)malloc(sizeof(struct _wayPoints));
                                cur=cur->next;
                                cur->x=obsX[j+1];
                                cur->y=obsY[j+1];
                                cur->rowNumber = lineNumber ;
                                cur->lineSegment = segmentNumber ;
                                cur->next = next ;
                          }
                    }

                    prev = cur;
                    cur = next->next;

                    while ((cur != NULL) && (cur->rowNumber == prev-
>rowNumber)){
                          if(obstacleRow[j]  == -2){
                                prev = cur->next;
                                cur = cur->next->next;
```

```
                        }
                    prev = cur;
                        cur = cur->next;
                }
            }
            obs = obs->next ;
      }
      fclose(fpMatlab);
      return (0);
}

/* decides if vehicle will travel clockwise or counter-clockwise
deepending on the
   distance from A to B.  Whichever is shortest will be taken */
int leftorRight(int count, int num_intersections,int num_vertices,
double obstacle_int_x[],
                        double obstacle_int_y[], int obstacle_int_ls[],
struct line_segment lsa[],
                        double boundryX[], double boundryY[])
{
      int start, end;
      int m,n;
      double distCW, distCCW;

      m = obstacle_int_ls[count];
      n = obstacle_int_ls[count+1];

      /* calculate distance clockwise from  first intersection point to
the second */
      start=m+1;
      end=n;

      if (start >= num_vertices)
            start = 0 ;
      if (end >= num_vertices)
            end = 0 ;
      distCW=sqrt(pow(lsa[start].x2-obstacle_int_x[count], 2)+
                  pow(lsa[start].y2-obstacle_int_y[count], 2));
      while(start!=end){
            if ((start+1) == num_vertices){
                  lsa[start+1].x1 = lsa[0].x1;
                  lsa[start+1].x2 = lsa[0].x2;
                  lsa[start+1].y1 = lsa[0].y1;
                  lsa[start+1].y2 = lsa[0].y2;
            }
            distCW+=sqrt(pow(lsa[start].x1-lsa[start+1].x1, 2)+
                  pow(lsa[start].y1-lsa[start+1].y1, 2));
            start+=1;
            if (start >= num_vertices)
                  start = 0 ;
      }
      distCW+=sqrt(pow(lsa[end].x1-obstacle_int_x[count+1], 2)+
                  pow(lsa[end].y1-obstacle_int_y[count+1], 2));

      /* calculate distance counter-clockwise */
      start=m;
      end=n+1;
```

```
	if (start >= num_vertices)
		start = 0 ;
	if (end >= num_vertices)
		end = 0 ;
	distCCW=sqrt(pow(lsa[start].x1-obstacle_int_x[count], 2)+
			pow(lsa[start].y1-obstacle_int_y[count], 2));

	while(start!=end){
		if ((start) == 0){
			lsa[start-1].x1 = lsa[num_vertices-1].x1;
			lsa[start-1].x2 = lsa[num_vertices-1].x2;
			lsa[start-1].y1 = lsa[num_vertices-1].y1;
			lsa[start-1].y2 = lsa[num_vertices-1].y2;
		}
		distCCW+=sqrt(pow(lsa[start].x2-lsa[start-1].x2, 2)+
			pow(lsa[start].y2-lsa[start-1].y2, 2));
		start-=1;
		if (start < 0)
			start = (num_vertices-1) ;
	}
	distCCW+=sqrt(pow(lsa[end].x2-obstacle_int_x[count+1], 2)+
			pow(lsa[end].y2-obstacle_int_y[count+1], 2));

	/* function returns 1 if vehicle turns clockwise or zero
otherwise */
	if(distCW<=distCCW)
		return(1);
	else
		return(0);
}

/* accounts for the turning radius of the vehicle */
int smoothWayPoints(float mroc, int numPathPts, struct _wayPoints
**path)
{
	void delete_path_pt(struct _wayPoints plan[], int i, int *count);
	struct _wayPoints *get_cubic_curve(struct _wayPoints *S, double
xe, double ye, double
		xf, double yf, double beta1, double beta2);


	int i,j,k,num_corner_sgs,skip_major_loop;
	double xc,yc,xd,yd,xe,ye,xf,yf,x1,y1,x2,y2,x3,y3,x4,y4;
	double gamma,theta1,theta2,rad,maga,magb,magn,crossprodm,dist;
	double xm,ym,s1,s2,beta1,beta2,tau,phe1,phe2,alpha,dbs;
	struct _wayPoints *wp, *plan, *tmp ;


	/* copy path to local variable array */
	plan = (struct _wayPoints *)malloc(sizeof(struct
_wayPoints)*numPathPts) ;

	for (i=0; i<numPathPts; i++)
	{
		plan[i].x    = (*path)->x ;
		plan[i].y     = (*path)->y ;
```

```
      plan[i].rowNumber   = (*path)->rowNumber ;
      plan[i].lineSegment = (*path)->lineSegment ;

      /* free memory used by *path */
      tmp = *path ;
      *path = (*path)->next ;
      free(tmp) ;
}

/* add first way point from plan */
*path = (struct _wayPoints *)malloc(sizeof(struct _wayPoints)) ;
wp = *path ;

wp->x        = plan[0].x ;
wp->y        = plan[0].y ;
wp->rowNumber      = plan[0].rowNumber ;
wp->lineSegment = plan[0].lineSegment ;
wp->next     = NULL ;


for (i=1; i<numPathPts-1; i++)
{
      x1=plan[i-1].x ;
      y1=plan[i-1].y ;
      x2=plan[i].x ;
      y2=plan[i].y ;
      x3=plan[i+1].x ;
      y3=plan[i+1].y ;

      theta1 = vec_angle(x3-x2,y3-y2,x1-x2,y1-y2)/2.0;
      skip_major_loop = FALSE;

      if(theta1 > 89.999*D2R)
      {
            skip_major_loop = TRUE;
            delete_path_pt(plan, i, &numPathPts);
            i--;
      }
      else if(i < numPathPts-2)
      {
            x4=plan[i+2].x ;
            y4=plan[i+2].y ;

            theta2 = vec_angle(x4-x3,y4-y3,x2-x3,y2-y3)/2.0;

            if(theta2 > 89.999*D2R)
            {
                  skip_major_loop = TRUE;
                  plan[i].x   = x1;
                  plan[i].y   = y1;
                  plan[i+1].x = x2;
                  plan[i+1].y = y2;
            }
            else
            {
                  s1 = mroc/tan(theta1);
                  s2 = mroc/tan(theta2);
```

```
                        magn = mag(x3-x2,y3-y2);
                        if(s1 > magn/2.0) /* 2-3 is a short segment*/
                        {
                              maga = mag(x2-x1,y2-y1);
                              magb = mag(x4-x3,y4-y3);
                              tau = vec_angle(x1-x2,y1-y2,x4-x3,y4-y3);
                              phe1 = ref_angle(x2-x1,y2-y1,x3-x2,y3-
y2);
                              phe2 = ref_angle(x3-x2,y3-y2,x4-x3,y4-
y3);
                              if(sign(phe1) != sign(phe2)) /* 1-2-3-4
is a zig-zag path*/
                              {
                                    skip_major_loop = TRUE;
                                    if(maga/2.0 < s1)
                                    {
                                          xe = (x2+x1)/2.0;
                                          ye = (y2+y1)/2.0;
                                    }
                                    else
                                    {
                                          xe = s1*(x1-x2)/maga + x2;
                                          ye = s1*(y1-y2)/maga + y2;
                                    }
                                    if(magb/2.0 < s2)
                                    {
                                          xf = (x4+x3)/2.0;
                                          yf = (y4+y3)/2.0;
                                    }
                                    else
                                    {
                                          xf = s2*(x4-x3)/magb + x3;
                                          yf = s2*(y4-y3)/magb + y3;
                                    }
                                    beta1 = ref_angle(1.0, 0.0, x2-x1,
y2-y1);
                                    beta2 = ref_angle(1.0, 0.0, x4-x3,
y4-y3);
                                    alpha = ref_angle(1.0, 0.0, xf-xe,
yf-ye);
                                    if((fabs(beta1-alpha) > 80.0*D2R)
|| (fabs(beta2-alpha) > 80.0*D2R))
                                          skip_major_loop = FALSE;
                                    else
                                    {
                                          skip_major_loop = TRUE;
                                          wp =
get_cubic_curve(wp,xe,ye,xf,yf,beta1,beta2);
                                          i++;
                                    }
                              }
                        }
                  }
            }

            if(!skip_major_loop)
```

```
            {
                  maga=mag(x1-x2,y1-y2);
                  magb=mag(x3-x2,y3-y2);
                  dist = mroc;
                  rad=mroc/sin(theta1);
                  magn = rad*cos(theta1);
                  dbs = DBS;
                  if((magn > maga/2.0) || (magn > magb/2.0))
                  {
                        magn = (maga < magb) ? maga/2.0 : magb/2.0;
                        rad  = magn/cos(theta1);
                        dist = rad*sin(theta1);
                        dbs  = DBS/2.0;
                  }
                  xc = rad*(x1-x2)/maga + x2;
                  yc = rad*(y1-y2)/maga + y2;
                  crossprodm=cross_product2(x1-x2,y1-y2,x3-x2,y3-y2);
                  if(crossprodm > 0.0)
                        k= -1;
                  else
                        k= 1;
                  vector_rotation(xc,yc,x2,y2,theta1,&xd,&yd,k);
                  perp_point(x1,y1,x2,y2,xd,yd,&xe,&ye);
                  perp_point(x3,y3,x2,y2,xd,yd,&xf,&yf);

                  gamma = M_PI/2.0 - theta1 ;
                  num_corner_sgs = (int)(2.0*gamma*dist/dbs + 0.5) - 1;

                  if(num_corner_sgs < 0)
                        num_corner_sgs = 0;
                  if(num_corner_sgs == 0)
                  {
                        wp->next = (struct _wayPoints
*)malloc(sizeof(struct _wayPoints)) ;
                        wp = wp->next ;

                        wp->x        = x2 ;
                        wp->y        = y2 ;
                        wp->rowNumber      = 0 ;
                        wp->lineSegment = 0 ;
                        wp->next     = NULL ;
                  }
                  else
                  {
                        for(j=1; j<=num_corner_sgs; j++)
                        {

      vector_rotation(xe,ye,xd,yd,2.0*((double)j)*gamma/

      ((double)(num_corner_sgs+1)),&xm,&ym,-k);

                              wp->next = (struct _wayPoints
*)malloc(sizeof(struct _wayPoints)) ;
                              wp = wp->next ;

                              wp->x        = xm ;
                              wp->y        = ym ;
```

```
                              wp->rowNumber      = 0 ;
                              wp->lineSegment = 0 ;
                              wp->next     = NULL ;
                         }
                    }
               }
          }
          if(mag(wp->x - plan[numPathPts-1].x, wp->y - plan[numPathPts-
1].y) > 0.001)
          {
               wp->next = (struct _wayPoints *)malloc(sizeof(struct
_wayPoints)) ;
               wp = wp->next ;

               wp->x        = plan[numPathPts-1].x ;
               wp->y        = plan[numPathPts-1].y ;
               wp->rowNumber      = 0 ;
               wp->lineSegment = 0 ;
               wp->next     = NULL ;
          }

          free(plan) ;
          return(0);
}

void delete_path_pt(struct _wayPoints plan[], int i, int *count)
{
  int j;

  for(j=i; j<*count; j++)
  {
      plan[j].x = plan[j+1].x ;
      plan[j].y = plan[j+1].y ;
  }

  *count--;
}

struct _wayPoints *get_cubic_curve(struct _wayPoints *S, double xe,
double ye, double
  xf, double yf, double beta1, double beta2)
{
  double A, B, C, D, det, c0[4], c1[4], c2[4], c3[4], y[4];
  double xpe, ype, xpf, ypf, s0, sf, tau;
  double xd, yd, xde, yde, xprev, yprev, tmp, magn;
  double xs, ys;

  det = 0.0;
  xs = xe;
  ys = ye;
  tau = ref_angle(1.0, 0.0, xf-xe, yf-ye);
  while(det == 0.0)
  {
    xy2xyprime(xe-xs,ye-ys,&xpe,&ype,tau);
    xy2xyprime(xf-xs,yf-ys,&xpf,&ypf,tau);

    tmp = beta1 - tau;
```

```
    if((cos(tmp) < 0.0) && (sin(tmp) > 0.0))
      tmp = 89.0*D2R;
    else if((cos(tmp) < 0.0) && (sin(tmp) < 0.0))
      tmp = -89.0*D2R;
    else if(sin(tmp) > 0.999)
      tmp = 89.0*D2R;
    else if(sin(tmp) < -0.999)
      tmp = -89.0*D2R;
    s0 = tan(tmp);

    tmp = beta2 - tau;
    if((cos(tmp) < 0.0) && (sin(tmp) > 0.0))
      tmp = 89.0*D2R;
    else if((cos(tmp) < 0.0) && (sin(tmp) < 0.0))
      tmp = -89.0*D2R;
    else if(sin(tmp) > 0.999)
      tmp = 89.0*D2R;
    else if(sin(tmp) < -0.999)
      tmp = -89.0*D2R;
    sf = tan(tmp);

    y[0]=ype; c0[0]=pow(xpe,3.0); c1[0]=pow(xpe,2.0); c2[0]=xpe;
c3[0]=1;
    y[1]=ypf; c0[1]=pow(xpf,3.0); c1[1]=pow(xpf,2.0); c2[1]=xpf;
c3[1]=1;
    y[2]=s0;  c0[2]=3.0*pow(xpe,2.0); c1[2]=2.0*xpe;  c2[2]=1;
c3[2]=0;
    y[3]=sf;  c0[3]=3.0*pow(xpf,2.0); c1[3]=2.0*xpf;  c2[3]=1;
c3[3]=0;

    det = det4(c0,c1,c2,c3);

    tau += (1.0*D2R);
  }
  tau -= (1.0*D2R);
  A   = det4( y, c1, c2, c3)/det;
  B   = det4(c0,  y, c2, c3)/det;
  C   = det4(c0, c1,  y, c3)/det;
  D   = det4(c0, c1, c2,  y)/det;

  xprev = xpe;
  yprev = ype;
  magn  = 0.0;
  for(xde=xpe; xde<=xpf; xde = xde + 0.01)
  {
    yde = A*pow(xde,3.0) + B*pow(xde,2.0) + C*xde + D;
    magn += mag(xde-xprev, yde-yprev);
    if(magn > 3.0)
    {
      xyprime2xy(xde,yde,&xd,&yd,tau);
      S->next = (struct _wayPoints *)malloc(sizeof(struct _wayPoints))
;
        S->next->x = xd+xs ;
        S->next->y = yd+ys ;
        S->next->rowNumber = 0 ;
        S->next->lineSegment = 0 ;
        S->next->next = NULL ;
```

```
        S = S->next ;
      magn = 0.0;
    }
    xprev = xde;
    yprev = yde;
  }

  return(S);
}

int polygonErrorChecking(double xpoly[], double ypoly[], int
num_vertices)
{
  int i;
  double sum_angle = 0.0;

  /* function prototype */
  int ccwPoly2cwPoly(double xpoly[], double ypoly[], int num_vertices)
;

  for(i=0; i < num_vertices; i++)
    {
      if(i == 0)
        sum_angle += ref_angle(xpoly[i]-xpoly[num_vertices-1],
ypoly[i]-
          ypoly[num_vertices-1], xpoly[i+1]-xpoly[i], ypoly[i+1]-
ypoly[i]);
      else if(i == num_vertices-1)
        sum_angle += ref_angle(xpoly[i]-xpoly[i-1], ypoly[i]-ypoly[i-
1],
          xpoly[0]-xpoly[i], ypoly[0]-ypoly[i]);
      else
        sum_angle += ref_angle(xpoly[i]-xpoly[i-1], ypoly[i]-ypoly[i-
1],
          xpoly[i+1]-xpoly[i], ypoly[i+1]-ypoly[i]);
    }

  if(fabs(-2.0*M_PI - sum_angle) < 0.0001)
    return(POLYGON_DEFINED_CORRECTLY);
  else if(fabs(2.0*M_PI - sum_angle) < 0.0001)
    {
      ccwPoly2cwPoly(xpoly, ypoly, num_vertices);
      //printf("\nERROR: polygon is defined CCW! Fixed locally\n");
      return(POLYGON_DEFINED_CCW);
    }
  else
    {
      printf("\nERROR: polygon intersects itself!\n");
      return(POLYGON_INTERSECTS_ITSELF);
    }
}

int
ccwPoly2cwPoly(double xpoly[], double ypoly[], int num_vertices)
{
  int i ;
  double tempX, tempY ;
```

```
  for (i=0; i<num_vertices/2; i++) {
    tempX = xpoly[i+1] ;
    tempY = ypoly[i+1] ;
    xpoly[i+1] = xpoly[num_vertices-1-i] ;
    ypoly[i+1] = ypoly[num_vertices-1-i] ;
    xpoly[num_vertices-1-i] = tempX ;
    ypoly[num_vertices-1-i] = tempY ;
  }
  return(0) ;
}
```

LIST OF REFERENCES

Arm00 Armstrong, D.G., Novick, D., Wit, J., Crane, C.D. A modular, scalable, architecture for unmanned vehicles. Center for Intelligent Machines and Robotics, University of Florida, Gainesville, Florida, USA.

Ran93 Rankin, A.L. 1993, "Path Planning and path execution software for an autonomous nonholonomic robot vehicle", Master's Thesis, University of Florida.

Row90a Rowe, N. C. 1990(December). Roads, rivers, and obstacles: optimal two-dimensional path planning around linear features for a mobile agent. *International Journal of Robotics Research, 9, no.6*, pp.67-74.

Row97 Rowe, N. C. 1997(June). Obtaining optimal mobile-robot paths with non-smooth anisotropic cost functions using qualitative-state reasoning. *International Journal of Robotics Research, 16, no.3*, pp. 375-399.

Row00 Rowe, N. C. and Alexander, R. S 2000(February). Finding optimal-path maps for path planning across weighted regions. *The International Journal of Robotics Research, 19, no.2*, pp. 83-95.

Row94 Rowe, N. C. and Kanayama, Y. 1994(October). Near-minimum-energy paths on a vertical-axis cone with anisotropic friction and gravity effects. *International Journal of Robotics Research, 13, no. 5*, pp. 408-432.

Row89 Rowe, N. C. and Lewis, D. H. 1989(January). Vehicle path planning in three dimensions using optics analogs for optimizing visibility and energy cost. *Proceedings of the NASA Conference on Space Telerobotics*, Pasadena, CA.

Row90b Rowe, N. C. and Richbourg, R. F. 1990(December). An efficient Snell's-law method for optimal-path planning across multiple two-dimensional irregular homogeneous-cost regions. *International Journal of Robotics Research, 9, no. 6*, pp. 48-66.

Row90c Rowe, N. C. and Ross, R. S. 1990(October). Optimal grid-free path planning across arbitrarily-contoured terrain with anisotropic friction and gravity effects. *IEEE Transactions on Robotics and Automation, 6, no. 5*, pp. 540-553.

Ste93 Stentz, A., Brumitt, B. L., Coulter, R. C., and Kelly, A. A system for autonomous cross-country navigation. *Proceedings of SPIE - The International Society for Optical Engineering v 1831 1993.* Published by Society of Photo- Optical Instrumentation Engineers, Bellingham, WA, USA, pp. 540-551.

BIOGRAPHICAL SKETCH

Vlad Hurezeanu was born on April 10, 1976, in Bucharest, Romania. After coming to the United States in 1994, he received the Associate of Arts degree from Broward Community College in 1997. In the same year he transferred to the University of Florida. He graduated with the Bachelor of Science in mechanical engineering with honors in August 1999. In the same year he enrolled in the master's program at the same university. After graduating from the master's program he will start working and pursue a Master of Business Administration concentrating on the engineering field.